Plan/Proof-of-Concept: 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 syntax, the *RSound* library functions utilized and the interpretation.

1. OVERVIEW

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 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 and the various features of the *RSound* library. 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. LOW RISK APPROACH

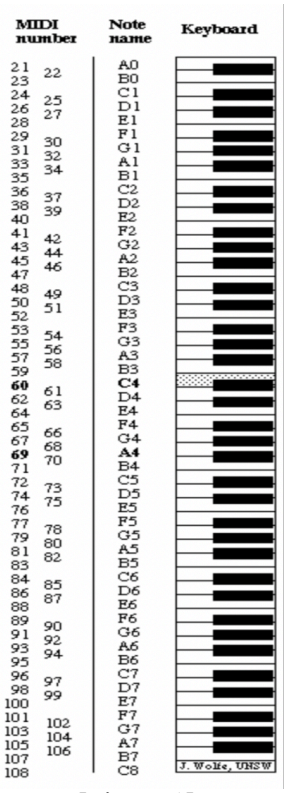
Our low risk approach for Loopy can be represented with two of our core goals: simplicity and functionality. The language must be simple enough for artists with no programming experiences to understand, and it need to function, as in create the music that the artist intended.

With these core goals in mind, we will explain three main components that will allow our proof of concept: the concrete syntax, RSound functions, and the interpretation.

3.1 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> ::=
| {note <num> <num> <num>}
| {loop <L-Expr>* <num> <num> <num>}
| {segment <L-Expr>* <num>}
| {<num> <L-Expr>}
```



[Figure 1]

The first 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 programming languages.

Figure 1 shows a conversion of a typical note and octave into the corresponding MIDI number[3]. The second and third parameters are the start and end bar in which the note will be playing for. A bar, in musical notation, is the segment of time

corresponding to a specific number of beats[4], and by default, all bars in Loopy will consist of 4 beats, or in musical notation, follow a time signature of $\frac{4}{4}$, meaning there will be 4 beats of quarter notes playing in one bar. All musical notation aside, to keep it simple to understand for the users we designed it so that all the user needs to provide is the start bar position and the end bar position for the synthetic musical note.

Our second form is the representative feature of our language which directly targets the repetitiveness of EDM: loops.

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

The parameters that a loop takes are: 1) a list of `LExpr` values to be looped, and like the note, 2) a start bar and 3) an end bar, and finally 4) 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 third 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. This validation will take place in the interpretation stage where the segment will actually clip the sound to the length of its total bars.

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, ends at the fourth bar, and repeats itself 4 times.

The first note will be playing a #50 MIDI note from the first bar until the fourth bar. The second note will be playing a #45 MIDI note from the first bar until the second bar. Finally, the last note will be playing a #30 MIDI note from the second bar until the second bar, so for just one bar.

The fourth and final 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[10], 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.

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

3.2 RSound

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

RSound[6] 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.

3.3 Interpretation

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

3.3.1 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 assembling with an empty sound. 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 returns frames of silent sounds. An example of an assemble for `{note 40 2 3}` would be as followed:

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

  (assemble
    (list
      (list (silence buffer) 0)
      (list
```

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

3.3.2 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:

When the 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.

When the type is `loop`:

Now, loops also have a start-bar and an end-bar. Similarly to notes, this can be handled with the prepending of silent sound buffers. That is all we need to do 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.

When the type is `segment`:

Segments do not have a starting bar, so appending to the assemble parameter list with a starting frame of 0 would suffice.

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 ebar 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)]
```

3.3.3 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.

3.3.4 Speed Modification

For speed modifiers, 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. HIGH RISK APPROACH

On top of our core goals, we have a few more ambitious goals to achieve for a full-on experience of Loopy: variable assignments and error handling. This will allow for even faster and more efficient EDM creation.

4.1 Variable Assignments

In Loopy, identifiers will be implemented through an environment that will be passed in with the recursive calls of the interpretator. This will allow us to:

a) Implement lexical scoping
b) Allow the end user to effectively access highly repeated musical structures

We plan to allow variable assignments in the following way:

```
{with {note-40 {note 40 1 4}}
      note-40}
```

The above code, specified by the end user, will play the note corresponding to `{note 40 1 4}`. In our implementation of identifiers and with statements we will only allow one binding in our with statements.

4.2 Error Handling

In Loopy, errors will be handled silently. We decided, based on or target user, that this would be the best implementation. Instead of failing to compile which could cause much frustration, errors will be caught and then the rest of the program will continue to play.

We have already talked about a design that we have built in silent error handling, namely, the segment. In the case where an end user gives a loop or other `L-expr` that has a end-bar which goes beyond the length of the total bar of the segment that it is placed in, the segment will actually 'silently' clip the sound by enforcing the total length to its own total bar. Basically, the erroneous `L-expr` will be silently cut off instead of it handling and throwing an error.

However, we do not have that for notes within loops. Since we recursively process the `L-expr`, if ever in one of the call stack an erroneous note is processed inside a loop, an error will be thrown. We would like to implement in such a way where that error is caught and handled by Loopy, rather than throwing an error.

5. SUMMARY

Our report has outlined the unique and useful features of Loopy that make it suitable for EDM development. By describing the concrete syntax, the *RSound* library functions that will be used and the 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 to fulfill our core goals and potentially our full goals.

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

[4] “Bar (Music).” Wikipedia, Wikimedia Foundation, 29 Apr. 2018, en.wikipedia.org/wiki/Bar_(music).

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

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