Final Project 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

Loopy is a domain specific language built for EDM (electronic dance music) creators, designed with solutions to provide simple approaches to the repetitive nature of EDM, such as with loops with various notes. This report will go through the core goals of describing the implemented concrete syntax, parser and interpreter with features like the *RSound* library used and error handling, while also describing the full goals of variable assignments and file export capabilities.

1. OVERVIEW

This report will go into detail into how we actually implemented and managed to create Loopy. We will begin with explaining all the features, categorized into core and full goals mentioned previously in our project plan. Then we will go into each feature and describe what it is about and how we came to create it.

To illustrate our work, we will include actual code snippets from our finalized project and outline the process in coming up with them with background information on the discussions our team had when creating them.

2. FEATURES

2.1 Core Goal Features

Most of our features come from our core goals of simplicity and functionality. The language must be simple enough for artists with no programming experiences to understand, and it needs to function, as in the ability to create the music that the artist intended. Most of the features that stem from our core goals revolve around the basic semantics of the Loopy language.

The following features are the basic components that allow our core goals:

- Concrete syntax
- Interpreter
- Error Handling

Of course, utilization of the *RSound* library, which is a monumental part of how our interpreter works will be explained as well to help understand how the interpreter came to be.

2.2 Full Goal Features

Our full goals also stem from our core goals of keeping it simple and functional for the users. Here are the full goal features:

- Variable Assignments
- Music Visualization
- Export capabilities

Variable assignments allow users to re-use notes, loops and segments in their music instead of writing them over and over again. This ensures a more efficient way of creating music.

Music visualization helps ambitious artists to analyze their music using sinusoidal wave functions. This will aid their music creation by visually showing how each note, loop or segment plays simultaneously or sequentially.

Finally, export capabilities will enhance the full experience of Loopy by providing a way for the users to extract their music into playable files. This will, in turn, allow users to either publish or share their music with whomever they wish.

3. THE LOOPY LANGUAGE

In this section, we will explain in detail of each feature, introduced in the previous section for both core and full goals, in their respective subsections.

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:

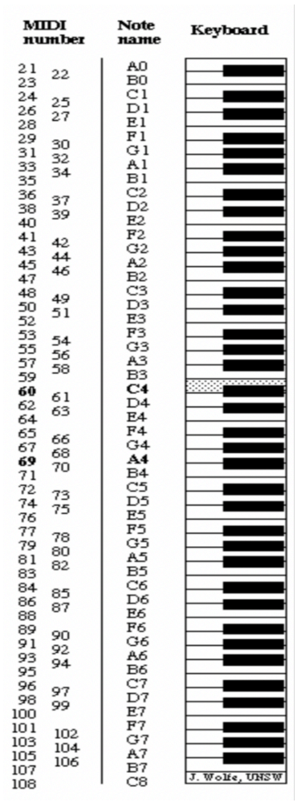
```
<L-Expr> ::=
| {note <num> <num> <num>}
| {loop <L-Expr>+ <num> <num> <num>}
| {segment <L-Expr>+ <num>}
| {<num> <L-Expr>}
| {with {{<id> <L-Expr>}*} <L-Expr>}
| <id>
```

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[1]. 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[2], 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.



[Figure 1: MIDI to Note Diagram]

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 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[3], 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[4], 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 fifth and sixth forms are a direct result of our effort to achieve variable assignments: *with* and *id.* We wanted users to basically have the capability to assign certain *L-Expr* objects to identifiers since in EDM reusability of beats and melodies are high. We will discuss about the detailed execution of how this comes about in section 3.3.

We can see that all the mentioned concrete syntax components align 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[5] 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 sound #10 as 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 simultaneously 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 note and loop implementations to concatenate sequentially 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

```
[note (mini-num start-bar end-bar)
 (define duration (- end-bar start-bar))
 (define validNote
  (if (negative? duration)
    false
    true))
 (define buffer
  (* (- start-bar 1) FRAME-RATE))
 (if validNote
  true
  (error 'Note "failed because note end-
             bar is before start-bar"))
 (if (zero? buffer)
  (synth-note "main"
             10
             midi-num
             (round (* duration
                       FRAME-RATE)))
  (rs-append
   (silence (round buffer))
   (synth-note "main"
              10
              midi-num
              (round (* duration
                        FRAME-RATE))))))]
```

The above code snippet is our interpretation on the note component. It first calculates three variables: *duration*, *validNote* and *buffer*.

The *duration* of the note is straightforward; we calculate the total duration, hence the total number of bars the note will be played, by simply subtracting the end-bar from the start-bar. In *validNote*, if this value comes out to a negative value, it will later be used to throw an error notifying the user that the note is malformed. Lastly, *buffer* is calculated simply by subtracting one from the start-bar, since for example if my start-bar was at 3, we want 2 silent bars filled for that note in the beginning.

We then want to return the *rsound*. The note that we will be playing is represented by the `synth-note`.

```
(synth-note
  "main"
  10
  mini-num
  (* FRAME-RATE duration))
```

This means that the `rsound` created gets our default sound (#10 sound 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.

But we cannot just simply play it; it depends on the buffer.

If the buffer is zero, it means that we do not in fact have to prepend any buffers. So we will simply just return the intended note formed by the `synth-note` function.

However, in the case it is not a zero, we will prepend silence by the amount of buffer we have calculated earlier. And to do that, we will be using the `silence` and `rs-append` function.

We will simply prepend the silent buffer, calculated earlier, onto the note created.

Why is this important? 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 prepending empty sounds. 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 `rs-append` stage is to create a buffer for the notes that have starting bars not at 1.

3.3.2 Loop

```
[loop (exprs start-bar end-bar iter)
 (local
  [(define duration (- end-bar start-bar)
   (define buffer (* (- start-bar 1)
                     FRAME-RATE))
   (define processed
    (assemble (map
     (lambda (n)(list n 0))
     (map
      (lambda (e) (helper e env))
      exprs))))
   (define loopAcc processed)
   (define (rec-append processed iter)
    (for ([i (sub1 iter)])
     (set! loopAcc
      (rs-append loopAcc processed)))
     loopAcc)]
 (if (negative? duration)
  (error 'Loop "failed because loop
          end-bar is before start-bar"))
  (if (zero? buffer)
   (rec-append processed iter)
   (rs-append
    (silence (round buffer)
    (rec-append processed iter))))]
```

The most crucial and important part of Loopy is the loop. This provides the main and basic function for tackling the repetitive nature of EDM.

Loops will have a similar approach to notes in terms of validation (Is end-bar larger than the start-bar?) and buffer prepending.

However, the major differences are in the way it handles the *L-Expr* objects that it has received and the iteration logic.

When a loop receives the list of expressions, it will use the `assemble` function to add up all the recursively processed *L-Expr* into one sound, whether it be a note, loop or segment. This is done by the *processed* variable where it goes through the list of expressions by mapping and recursively passing each expression into the *interp helper* function.

Once *processed* is complete, the local function *rec-append* will iteratively append the retrieved processed *rsound* by the number specified by Loop's own *iter* parameter. This signifies the actual looping nature.

Then, like *note*, it will prepend any buffer it has and return the complete *rsound*.

3.3.3 Segment

```
[segment (exprs total-length)
 (local [
  (define processed
    (assemble (map
     (lambda (n)(list n 0))
     (map
      (lambda (e) (helper e env))
      exprs))))]
 (clip processed 1 (* FRAME-RATE
                      total-length)))]
```

The segment is simply a structure that defines the length of whichever `L-Expr` objects 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 by the number of total bars that it was given.

3.3.4 Speed Modification

```
[modify-speed (multiplier expr)
 (resample multiplier (helper expr env))]
```

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.

3.4 Error Handling

In Loopy, there are two kinds of errors: explicit and implicit.

Explicit error are the ones we throw to let the user know that there's no way we can fix what the user did so please fix it. For example, in our *note* and *loop* components, we explicitly throw an out-of-order bar error when the end-bar parameter seems to be larger than the start-bar parameter. This is crucial since in Loopy, without the correct bar positions the notes or loops will simply fail to play.

Then, there are the implicit errors, which are silent. We decided that instead of explicitly throwing errors, for some of Loopy that we can actually handle user error.

A example is the design of segments. 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.

By utilizing both explicit and implicit errors we, again, try to give users the best experience.

3.5 Variable Assignments

Variable assignments were the first feature of our full goals for this project; we wanted what was best for our users.

This was indeed evident from our concrete syntax as we introduced the *with* and *id* components so that users can specify which sounds were set to which identifiers so that they could re-use the sounds later on and allow them to effectively access highly repeated musical structures.

This was done in the interpreter by introducing environments.

```
;; The environment that stores
;; "deferred subs".
(define-type Env
 [mtEnv]
 [anEnv (name symbol?)
        (val rsound?)
        (restEnv Env?)])

;; lookup: symbol Env -> rsound
;; Find the value of name in env.
;; Emit error if not found.
(define (lookup name env)
 (local [
  (define (lookup-helper name
                         env)
   (type-case Env env
    [mtEnv () (error 'lookup "free
                    identifier: ~a" name)]
    [anEnv (bound-name bound-vl rest-env)
     (if (symbol=? bound-name name)
         bound-vl
         (lookup-helper name rest-env))]
     ))]
 (lookup-helper name env)))
```

The two functions above creates the environment where the interpreter stores user-defined sounds to its identifier. Whenever an identifier is interpreted it will lookup the name of the identifier in the environment and retrieve what is needed.

As for the with component, it will store in the recursively passed environment the identifier and the sound that corresponds to it.

This is how variable assignments is implemented within Loopy.

3.6 Music Visualization

Music visualization is not something we implemented so the following is a plan for future implementation. We will use the following method from the the *RSound* library:

```
(rs-draw rsound) -> void?
```

The above code creates a visualization of an *rsound* with a sinusoidal wave function. As our music is interpreted into a single *rsound* we will invoke the *rs-draw* function with the single *rsound*. This will allow the user to visualize anything from a single note to a complete song. We plan to allow for the option of a visualized sound by adding a parameter to each segment. This parameter will be a boolean value (0 or 1) that will specify if the sounds inside will be visualized. In our *interp* function we will simply check if the value is true at the end of the interpretation and call the above method.

3.7 Export Capabilities

We plan to to this using the following methods from the *RSound* library:

```
(rs-write rsound
          path) -> void?
```

The code shown above allows an *rsound* to be written to a wav file specified by the path parameter. Along with the music, the visualization will be written to multiple image files as well. This will be done with the htdp/image library. We will use the following method to save multiple image files to disk:

```
(save-image image
            filename) -> boolean?
  image : image?
  filename : path-string?
```

Similarly to the code we will use for music visualization, the parameter for exporting will be added to segment. At the end of the interpretation function we will call the above methods to write the final *rsound* to disk. The same will be done for the images generated from the *rs-draw* method. We plan to include two parameters; one for saving the images and one for saving the final *rsound*.

4. SUMMARY

Our report has outlined the unique and useful features of Loopy that make it suitable for EDM development. Our core goal lays the foundation for the concrete syntax, interpreter and the error handlings necessary for the basic functionality and simplicity that we envision for our users. Our full goal then extends that into an useful and enhancing experience by adding variable assignments, music visualization, and export capabilities. This is what we have visioned for Loopy.

REFERENCES

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

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

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

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

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