

Team Members

Ziyu Wang(wangz59), Wei Jiang(kidsbear), Xinyu Sui(suix2)

Program

```
function <p8>(x) = (x,/x);
function <arpeg2>(x, y) = (-xy);
function <main>(1) = <arpeg2><p8>(12);
```

Programs are formed by function definitions.

Functions bodies are formed by *Elements*(with some sugars) and function applications.

Implementation

We will use ANTLR to parse the program into an AST. We will apply desugaring to some grammar, while AST optimization is low-priority for us. We will write an *interpreter* to interpret this AST into a structure that can be easily converted into MIDI. Finally, we will write a *performer* to convert this structure into a MIDI file. The interpreter and performer will be written in Javascript.

Implementation Steps (Milestones)

These are the milestones for this project that are decent for demo after finishing them. Refer to the whole list of tasks in “Tasks” section.

1. Grammar parser and transformation into *Element* structures, without sugars and functions
2. Performer(*Elements* into MIDI converter)
3. Interpreter - Function application logic
4. Grammar desugaring (Translate prefix/infix/suffix into function)
5. Custom function definition
6. UI

Interpreter

At runtime, we keep a list of current *elements*. When we exit a function node, we apply the function to each element in this list.

For example, a program:

```
function <p8>(x) = (x,/x);
function <arpeg2>(x, y) = (-xy);
function <main>(1) = <arpeg2><p8>(12);
```

First we parse the program into a tree. When we walk off the node for “123”, we have

```
[Note(1/4,1), Note(1/4,2)] // Note(length, pitch)
```

After we apply the function <p8>:

```
[Closure(Note(1/4,1), Note(1/4,/1)), Closure(Note(1/4,2), Note(1/4,/2))]
```

After we apply function <arpeg2>:

```
[Note(1/8,1), Note(1/8,/1), Note(1/8,2), Note(1/8,/2)]
```

Finally we send these into the Performer to translate into MIDI file.

Elements

Elements are both structures to be modified by functions while interpreting and can be easily translated into a MIDI file by the Performer.

Class name	Fields	Description
Element	content - Note Wrapper Closure	
Note (Element)	duration - fraction pitch - float	
Closure (Element)	size - int elements - List<Element> * size	List of grouped elements wrapped by (). We will use X-closure to denote closure of length X, like 1-closure, 2-closure, etc. We will call this.elements[i] “the i’t h fragment”.
Wrapper (Element)	elements - List<Element>	Elements wrapped by []. Elements inside is treated as an integrated element. For example, with function <p8>(x) = (x, /x), <p8>([123]) = ([123], [/1/2/3]), while <p8>(123) = (1,/1)(2,/2)(3,/3). If an n-Closure is left inside a Wrapper, it will be regarded as parallel notes while converting to MIDI file.

Functions

Functions are defined for Notes to modify pitch, duration or even split into several notes.

Functions are applied to a list of elements. Each element is applied with this function and individual results are concatenated to form the result list. (If the result is a 1-closure, it will be broken into elements inside)

If the element is a Closure or Wrapper, it may have special effects, for example, see **@Dispatched** below.

Input/Output type: functions may take a x-closure and return a y-closure, x and y can be dynamic.

Core functions are listed below:

Function Name	Input/Output type	Description
<main>	1-1 n-n	Must be defined by user. Although it has “n-n” in its type, it will not be used unless user uses it(highly unlikely).
<get@x>	any-1	Gets the (x mod y)’th part from a y-closure. If x > y, warn(“Get-Trimmed”).

<check@x>	any-x	Transforms every element into x-closure with contents inside. Notes and Wrappers are casted into 1-closure, and then for each y-closure element, the result's i'th fragment will be its (i mod y)'s fragment. Result is fused into a single x-closure. If $x < y$ for any y, warn("Check-Trimmed"). If $x \bmod y \neq 0$ for any y, warn("Check-NonDivisible")
<repeat@x>	n-1	@Checked(x) Flattens into a 1-closure. For $i = 0$ to x, concat all elements' i'th fragment. After that, concat all x results. Generated from (...)*
<parallel>	n-1	Returns the Closure wrapped by a Wrapper. This way the closure will never be checked and will be regarded as parallel. See "Wrapper" above. Generated from (... , ... , ...)^
<customName> with x-closure argument and returns y-closure	x-y n-n	@Dispatched @Checked(x) For now, we will assume that all user-defined functions have the above features. We may change it if it turns out to be not ideal, for example, change @Checked(x) into @IgnoreUnless(x).
<@x/@y>	n-n	@Dispatched Multiplies the note length by x/y.
<&x-&y>	n-n	@Dispatched Modifies the pitch of note by difference between note y and note x.

@Dispatched: For closures and wrappers, the function is applied to all its elements inside.

@Checked(x): Applies <checkx> beforehand.

@IgnoreUnless(x): If the element length matches x, then apply, otherwise keep original.

Grammar

```

/* ----- Building blocks ----- */
// Notes
noteName: [ac-zA-Z0-9] // Legal names for notes
prefix: [#b/\] // Prefixes that can be applied to raw notes
suffix: [-.] // Suffixes that can be applied to raw notes
ignore: |

// Closure
startClosure: (
endClosure: )
closureInfix: + | -
closureSuffix: [*^]
closureSeparator: ,

// Wrapper
startWrapper: [
endWrapper: ]

```

```
// Function
functionName: [a-zA-Z0-9+~]*
startFunction: <
endFunction: >
functionDefKeyword: function
functionDefAssign: =
```

```
// Pattern match
startPattern: #startClosure
endPattern: #endClosure
patternSeparator: #closureSeparator
```

```
// Math
positiveInt: [1-9][0-9]*
```

```
/* ----- Elements ----- */
```

Prefixed notes: they are of a certain pitch. Duration is undefined.
 Notes: they have defined pitch and duration.

```
element: #note | #closure | #warpper
elements: #element+
```

```
prefixedNote: #prefix* #noteName
note: #prefixedNote #suffix*
```

```
rawClosure: (#closureInfix #fragment (, #fragment)*)
closure: #function* #rawClosure #closureSuffix*
```

```
wrapper: #startWrapper elements #endWrapper
```

```
/* ----- Functions ----- */
```

Function: see Functions

```
function: <#functionName>
```

FunctionDef: Defines a no argument function.
 PatternMatcher: Matches a closure of some length and give names to fragments of each index.

```
functionDef: function #noArgFunction #patternMatcher = #closure
patternMatcher: (#noteName (, #noteName)*)
```

Program: a collection of function definitions. Main part will be written like a 1-1|n-n
 function: function <main>(1) = 12345--- (with 2 as an alias of ##1, etc)

```
program: functionDef+
```

Tasks

- Set up development environment

- ANTLR grammar parser
 - Grammar processing
 - Grammar wiring / desugaring
 - Note prefixes #1 b1 /1 \1
 - Note suffixes 1-- 1.
 - Closure infix(+ ...) (- ...)
 - Closure suffixes (...)* (...)^
 - Special functions
 - Length <1/2>
 - Transpose <1-2>
 - Pattern matcher (x, y)
 - Track pattern length
 - Function definition
 - Interpreter - AST logic
 - Closure
 - Function applied to Closure
 - Wrapper
 - Function applied to Wrapper
 - Function definition
 - ...
 - ...
 - Built-in functions
 - Fraction number interface
 - Performer - MIDI converting
 - UI
 - Colored code
 - Formatter
 - Play/Convert button
 - Playing note highlight
-