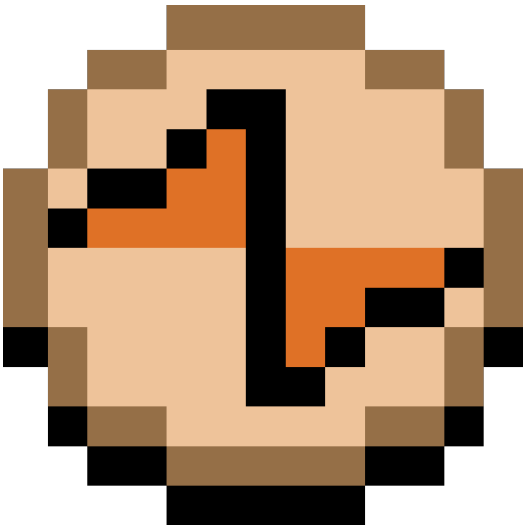

Decibel

Aditya Garg - CS22BTECH11002

Anirudh Saikrishnan - CS22BTECH11004

Bonthu Mani Hemanth Reddy - CS22BTECH11013

Edward Nathan Varghese - CS22BTECH11020

Kaipa Venkata Tuhil - CS22BTECH11030

Shaik Police Parvez - CS22BTECH11052

Syed Abrar - CS22BTECH11058

September 2024

Contents

1 Introduction

Decibel is a simple, intuitive, statically typed, domain-specific audio programming language designed to simplify audio generation and processing.

1.1 Background

The need for high quality audio for films, music and video game production is growing at an immense rate. Audio programming languages have been designed since the late 1980s however these tend to be oriented towards domain experts. This necessitates the need for specialised yet intuitive programming language to handle complex audio operations with ease. The existing DSLs and tools catering to this domain provide a useful way to interact and modify audio files but present a challenge to the user with a steep learning curve. This gives rise to the need for a DSL such as Decibel which improves the syntax and user productivity, without compromising on the efficiency and complexity of the audio operations.

1.2 Aim

The aim is to provide a powerful yet simple DSL for audio programming. With decibel, you can take multiple audio files and combine them in many ways, with a wide range of operators, including concatenation, superposition, slicing, distortion, high pass filter, low pass filter, and panning. With these operations, one can easily manipulate audio files to their liking. Another important part of decibel is audio generation. This can be used to generate music and sound effects. We also want decibel to borrow some aspects of functional programming, so we provide an easy and powerful syntax for currying. Apart form these aspects, we also aim to provide cross operator optimisations to improve the performance of these audio operations.

2 Language Specifications

2.1 Data Types

Decibel provides the following primitive data types that are generally available in other languages. They are **int**, **long**, **float**, **bool**, **audio** and **string**.

2.1.1 Integers

Integer numeric data types are 16-bit **int** and 64-bit **long**. There is no unsigned integer data type in Decibel. When you declare an integer data type without explicit type, the compiler tries to fit using int. If it's not possible, it uses long.

2.1.2 Float

For real numbers, Decibel provides a 64-bit **float** data type. It follows the IEEE 754 double precision standard.

2.1.3 Boolean

A **bool** data type can take either of two values: true or false. Any non-zero value is considered true and zero is considered as false during assignment/declaration.

2.1.4 Audio

Decibel supports **audio** data type. It is an audio file which can be manipulated using in-built operators. It can be saved as a .mp3 or .wav file.

2.1.5 String

String is a sequence of characters enclosed in double quotes (“). In decibel, strings are immutable.

2.2 Literals

1. **Integer**: An integer literal contains only decimal digits(0-9). It can optionally have the sign(+, -) as prefix. It cannot contain decimal points. It can be assigned to any data type except Audio and String.
2. **Float**: A float literal contains decimal digits(0-9) and at most one decimal point. It can optionally have the sign(+, -) as prefix. It can only be assigned to the float data type.

   A float literal can optionally have a case insensitive suffix of s, ms, hz. Literal with ms as suffix will be divided by 1000 whenever it is used and literal with hz as suffix will be converted into semitones.
3. **Bool**: There are only two bool literals: true and false. It can only be assigned to a bool data type.
4. **String**: A string literal contains a sequence of characters enclosed in double quotes(“). It can only be assigned to string data types.

2.2.1 Identifiers

Identifiers are used to name variables, functions. In Decibel, an identifier is a sequence of characters. The first character can be any alphabet(any case). The subsequent characters can be underscore, alphabet(any case) or a digit(0-9).

2.2.2 Keywords

The following keywords are reserved by Decibel and cannot be used as identifiers or functions.

`const`	`load`	`save`	`play`	`function`
`if`	`or`	`otherwise`	`loop`	`over`
`long`	`int`	`float`	`string`	`audio`
`bool`	`true`	`false`	`continue`	`break`
`return`	`HIGHPASS`	`LOWPASS`	`EQ`	`SIN`
`COS`	`EXP_DECAY`	`LIN_DECAY`	`SQUARE`	`SAW`
`TRIANGLE`	`PAN`	`import`	`to`	

Table 1: Reserved Keywords in Decibel

2.3 Operators

Decibel aims to make fundamental audio operations possible using the most intuitive operators for each of them. In addition to this, it supports most standard operators for integers and floats as well.

2.3.1 Arithmetic Operators

- $'+'$: Addition of two variables. Unless explicitly typecast, all variables will implicitly be cast to the type with the highest precedence in the current expression. The precedence order is as follows:

  $$String > Float > Long > Int$$

  If there is a String variable in the expression, the '+' essentially means concatenation

- $'-'$: Subtraction
  $(a - b) =>$ b is subtracted from a

- $'*'$: Multiplication of two variables

- $'/'$: Division
  $(a/b) =>$ a is divided by b. If either a or b are not floats, the integer part of the division is returned.

- $'\%'$: Remainder
  $(a\%b) =>$ The remainder when a is divided b. Note that neither a, nor b can be floats.

- $'\hat{}'$: Exponentiation
  $(a\hat{}b) =>$ The number a is raised to the power of b

Note: None of the arithmetic operators other than '+' apply to strings

2.3.2 Logical Operators

- $'\&\&'$: Logical AND

- $'||'$: Logical OR

- $'!'$: Logical NOT

Note: The above operators work exactly like their C/C++ counterparts. They act on two expressions(1 in the case of !) and return a boolean value.

2.3.3 Relational Operators

- $' <'$: Less Than

- $' <='$: Less Than or Equal To

- $' >'$: Greater Than

- $' >='$: Greater Than or Equal To

- $' =='$: Equal To

- $'! ='$: Not Equal To

Note: The above operators also work like their C/C++ counterparts.

2.3.4 Assignment Operators

The following assignment operators also work like their C/C++ counterparts:

$$=,\ +=,\ -=,\ *=,\ /=,\ \%=\ |=,\ \hat{}=,\ \&=$$

The following two new operators are also introduced:

1. <-: Used to declare a new variable and assign a value to it. All new variables should be declared using this operator before assigning a value to them using '=' or similar operators.
2. ->: Used to save an audio variable to an actual audio file.

2.3.5 Audio Operators

- *Concatenation:*
  Two audio files a1 and a2 can be concatenated using the '+' operator. The expression a1+a2 implies that the file a2 is appended to the back of file a1.

- *Slicing a File:*
  Decibel allows users to extract specific portions of an audio file based on certain parameters. Since an audio file is stored as an array of values, the '[ ]' operators can be used to access those specific portions.
  Syntax:<Audio File>[Start : Stop]
  Both Start and Stop are floating point integers and represent the start and end points within the audio file in seconds.
  Eg:- b = a[10: 25] will slice the portion of the file a starting at 10s and ending at 25s, and stores it in b.
  Note: If Start is less than 0 or if Stop is greater than the actual length of the audio file, the extra time will be padded with 0s (no sound).

- *Repeating a File*:
  A particular file 'a' can be repeated multiple times using the '^' operator followed by the number of times you want the file to be repeated.

  - a^n where n is an integer: The file a will be repeated n times
  - a^(n) where n is a floating point integer: For n1 and n2 such that n = n1+n2 where n1 is the integral part of n and n2 is the fractional part of n, the file will be repeated n1 times and n2 fraction of the file will then be appended to the back of the repeated segment

  Eg:

  1. a^5 will repeat the file 'a' 5 times
  2. a^7.8 will repeat the file 'a' 7 times and then append the starting 0.8 fraction of 'a' to the back

- *Superposition:*
  Two audio files a1 and a2 can be superimposed onto one another using the '|' operator. If one of the files is larger than the other, the returned file will have the size of the larger file, where the last part where they don't overlap will contain audio only from the larger file.

- *Scaling:*
  An audio file can be scaled using the '*' operator. This scaling can happen across the file by a constant factor or by a function that scales different portions of the file by different amounts.

  - a*z where z is an integer/float: The amplitude of every point of the wave will be scaled by a factor of z.
  - a*f where f is a function: The amplitude of every point of the wave will be scaled according to the function value at that particular instant in time. f must be a function which takes in the time value as a float and returns a float.

- *Speed up/Slow down:* An audio file can be sped up/slowed down using the '>>' and '<<' operators respectively.
  - a>>n will speed up the file by a factor of n - This happens by compressing the file to 1/n times its original size.
  - $a << n$ will slow down the file by a factor of n - This happens by expanding the file to n times its original size.

- *Complement of a File:*
  The complement of an audio file can be extracted from a file using the '~' operator. Every element in the audio array will be negated to generate the required complement.

- *High Pass Filter:*
  Frequencies below a certain value can be filtered out of an audio file using the High Pass Filter.
  Syntax: HIGHPASS(<Audio variable>a, <Floating point literal>f)
  OR HIGHPASS(<Audio variable>a, <Function>f)
  Case 1: 'a' is the audio file and 'f' is a constant frequency below which all frequencies will be filtered out
  Case 2: 'a' is the audio file and 'f' is a function that specifies the threshold value for every instant in time. The frequency at any point in the file will be retained/filtered out based on the function value at that point.

- *Low Pass Filter:*
  Similar to a high pass filter, Low Pass filters out higher frequencies while allowing lower frequencies to pass through
  Syntax: LOWPASS(<Audio variable>a, <Floating point literal>f)
  OR LOWPASS(<Audio variable>a, <Function>f) Same cases as High Pass Filter

- *Equaliser:*
  This will fit the sound to a given equaliser function. This equaliser function must take in the frequency in semitones, and returns the scaling factor, a float.
  Syntax: EQ(<Audio variable>, <Equaliser function>)
  This will return the equalised audio.
  This is useful, for example, if you want to remove certain frequencies from the sound or increase or decrease the bass.

- *Distortion:*
  At every point of time in the audio file, replace the value at that point by the value returned by a function that you pass in.
  Syntax: <audio> & <distortion function>
  Distortion function must take an int as input and output an int.

- *Panning:*
  Make the audio panned to the left or right ear, or somewhere in between.
  Syntax:

```
PAN(<Audio variable>, <value>)
```

Here, <value> can be either a float literal or a time dependent function. A value of -1 represents completely panning to the left, 1 means completely panned to the right, and 0 means no panning. If a value does not lie in this range, it will be clamped to this range. This returns an audio file.

2.3.6 Operator Precedence

Precedence	**Operator(s)**	**Associativity**
1	[...] **slice operator**	Left-to-right
2	~, !	Right-to-left
	HIGHPASS, LOWPASS	Left-to-right
3	^	Left-to-right
4	&	Left-to-right
5	$*$, /, %, >>, <<, PAN	Left-to-right
6	+, −, |	Left-to-right
7	<, <=, >, >=, ==, ! =	Left-to-right
8	&&, ||	Left-to-right
9	=, +=, -=, *=, /=, %=, |=, ^=, &=, <-, ->	Right-to-left

Table 2: Operators and Associativity

2.3.7 Implicit Casting

Implicit type casting from $bool- > int- > long- > float$ is supported for all the logical, arithmetic and assignment operators.

2.4 Comments

Decibel follows C++ style syntax for comments. Nested comments are not allowed in Decibel.

2.4.1 Single Line Comments

Single Line Comments begin with `//` and extend to the end of the line.

2.4.2 Multi Line Comments

Multi Line Comments start with `/*` and end with `*/`.

2.5 Statements

2.5.1 Import Statement

This is used to import another decibel program into current program. It import all the global variables and functions(except main function).

```
import <fileName>;
import "mainProgram.db";
```

2.5.2 Declaration Statements

1. **Variable Declaration and Initialization**: This is used to declare and initialize a variable of a specified type.

   ```
   identifier: type <- initializer;

   duration: float <- 5.0;
   isActive: bool <- true;
   filename: string <- "track1.wav";
   ```

   However, if they type can be inferred from the right hand side, there is no need to specify the type of the variables, and it will be infered automatically.

   ```
   duration <- 5.0; // duration is a float
   isActive <- true; // isActive is a boolean
   filename <- "track1.wav"; // filename is a string
   ```

2. **Constant Declaration**: This is used to declare and initialize an unchangeable variable.

   ```
   const identifier: type <- initializer;
   const maxRetries: int <- 5;
   const silence: audio <- load "silence.wav";
   ```

3. **Audio Declaration**: This is used to declare an audio file variable and optionally load it.

   ```
   identifier: audio <- load "filename";

   backgroundMusic: audio <- load "background.mp3";
   ```

2.5.3 Assignment Statements

Decibel provides various assignment statements. Here are a few along with appropriate syntax.

1. **Basic Assignment**

   ```
   identifier = expression;
   count = 10  // Example Usage
   ```

2. **Assignment with Audio File**

   ```
   audio_variable = load "filename";
   track = load "track1.wav";  // Example Usage
   ```

Similarly, operators like += and similar ones that involve other operators can be used in the same way as long as the operation between the variable to the left of these operators and the expression to their right is allowed and valid.

2.5.4 Audio Statements

Decibel supports playing and saving an audio file.

1. **Play Statement**: It plays the specified audio file.

```
play audio_variable;
play track1;
```

2. **Save Statement**: It saves the audio variable to the specified file.

```
save audio_variable -> "filename";
save track1 -> "output.wav";
```

2.6 Control Flow

Control Flow refers to the order in which individual statements, instructions, or function calls are executed in a program. In Decibel, conditionals handle decision-making based on expressions, while loops facilitate repetitive execution of code blocks based on conditions or ranges.

2.6.1 Conditionals

Decibel provides support for conditional statements using the ‘if‘, ‘or‘, and ‘otherwise‘ structure. Conditionals are enclosed in curly braces to define their scope, which avoids shift-reduce conflicts during parsing. Since the curly braces describe the scope of the statements, Decibel does not necessitate the use of the ‘or‘ and ‘otherwise‘ statements.

2.6.2 Syntax

```
if <expression> {
    <statements>
} or <expression> {
    <statements>
} ...
otherwise {
    <statements>
}
```

2.6.3 Examples

```
if duration > 10 {
    play track1;
} or duration > 0 {
    save track1 -> "output.wav";
} otherwise {
    audio track1 <- load "input.wav";
}
```

2.6.4 Loops

Loops in Decibel allow for iteration based on a multiplier or over a specified range. Two types of loops are supported: fixed iteration (‘loop times‘) and range iteration (‘loop over‘).

- The ’loop times’ statement executes the statements within the loop ‘times‘ number of times.
- The ‘loop over‘ statement loops over an identifier from start to end (non-inclusive) with a step of step_integer. It is not mandatory to specify the step value, as its default is considered as 1.

2.6.5 Syntax

```
loop <times> {
    <statements>
}

loop over <identifier> <start> to <end> [@<step_integer>] {
    <statements>
}
```

2.6.6 Examples

```
loop 3 {
    sound1 <- load "sound1.wav";
    play sound1;
}

loop over i 1 to 5 @2 {
    sound2 <- load "sound2.wav";
    play sound2;
}
```

2.7 Functions

Decibel provides functionality for defining functions. Users can create their own functions and call them anywhere in the program. In addition to the standard features, Decibel supports **currying**.

2.7.1 Definition

Functions in Decibel are declared using the `function` keyword followed by its name. Every function must be defined at the time of its declaration. The data type of arguments must be specified. The return type should be explicitly declared. A return type can be omitted if and only if the function does not return a value. Functions can also be written as inline expressions.

2.7.2 Syntax

```
function <function name> <- ([<parameter>:<type>,...]):<return type>{
    <function body>
}
```

(or)

```
function <name> <- ([<parameter>:<type>,...]):<return type> => <Expression>;
```

2.7.3 Examples

```
function add<- (a: int,  b: int): int {
    return a+b;
}
function add<- (a: int, b:int ): int  => (a+b);
```

2.7.4 Calling

A function can be called by its name followed by parentheses containing the parameters in the correct order.

```
result <- add(2, 3);
```

2.7.5 Currying

Currying is a powerful technique of transforming a function that takes multiple parameters into sequence of families of functions that take fewer arguments. Decibel provides simple and powerful syntax for currying.

When you call a function with one or more parameters omitted, Decibel returns a new function which takes omitted parameters as input and given parameters are inbuilt into the function. Parameters can be omitted by passing **underscore(_)** in place of the parameter during function call.

```
function add2 <- add(2, _);
value <- add2(5)    // Equivalent to add(2, 5)
```

If the omitted parameters are the last parameters, it is optional to use underscore for them. You can omit as many parameters as you want and the resulting function will accept those parameters in the order they were omitted.

```
function addTwo <- add(2)   // Equivalent to add2
```

2.7.6 Usage with Operators

Functions can be used along with operators. For instance, you can multiply an audio file with a function that takes the value at a specific point of time.

2.8 Audio Generation

Decibel supports audio generation using its `audio` function with the following syntax.

```
audio(<Wave Function>, <Frequency in semitones>, <Length>)
```

It returns an audio variable if called with all the parameters, or like any other function, returns a function which takes in the omitted parameters.

2.8.1 Wave Function

The `wave function` must be a function that takes a float in the range (0, 1) as parameter and returns an integer. It is used to calculate the waveform based on the specified frequency. Decibel provides the following wave functions by default: `SIN`, `COS`, `SQUARE`, `SAW` and `TRIANGLE`. Users can define their custom wave functions.

2.8.2 Frequency in semitones

The `frequency` can be an int, long, float, or a function that takes a float as parameter and returns a float. The value of the frequency must be in semitones, with a value of 0 representing the middle C in music, ie., 440hz. You can also pass in a frequency literal like `440hz`. Every 12 semitones is an octave. If an int or long is passed, it will be internally typecast to a float. This frequency determines the time period of the resulting waveform. If a function is passed as the frequency, it will be used to calculate the frequency as a function of time.

2.8.3 Length

The `length` specifies the duration of the audio in seconds. This parameter must be a float (or an integer, but it will be implicitly typecast in that case).

2.8.4 Examples

Example 1:

```
wave1 <- audio(SIN, 440hz, 5s);
```

Here, we are calling the audio() function with a SIN waveform, with a frequency of 440Hz and a length of 5s. This will generate a sin wave at middle C, which will last for 5 seconds when played.

Example 2:

```
function linear_decay_1s <- (time: float):float {
    if time < 1{
        return 24 - time*24;
    } otherwise {
        return 0;
    }
}

wave2 <- audio(SAW, linear_decay_1s, 2s);
```

This will generate a SAW wave which initially starts out at 24 semitones (2 octaves) above middle C, linearly slides down to middle C over the period of 1 second and remains there for another second.

3 Program Structure

A decibel program consists of function definitions and declarations. There must exist exactly 1 function with the name "main", which takes no input parameters and has no return value. All the declarations outside the main function must only be initialised with compile time constants. The main function can then contain declarations, assignments and control flow statements.

```
import "myProgram1.db";

const fileName: string <- "myMusic.mp3";

function add<- (a: int, b: int): int {
```

```
    return a+b;
}

function plusplus<- add(1);

function main<- () {
    // Declaration Statement
    audioFile: audio <- load fileName;
    newFile: audio <- 0;
    a: int <- plusplus(2);

    // Conditional Statement
    if a%2==0 {
        newFile = audioFile >> 2;
    } or a==1 {
        newFile = audioFile * 3;
    } otherwise {
        newFile = audioFile ^ 5;
    }

    /* This is a
       Multi Line Comment */

    play newFile;

    // Saving the file
    save newFile -> "newMusic.wav";
}
```

4 Optimisations

The following but not limited are some potential optimizations that could be implemented to improve the overall performance of the compiler.

4.1 Cross Operator Optimisations

Many sequences of complex operations on audio files can be simplified to equivalent simpler operations or functions. The use of multiple operators can be computationally inefficient and takes more memory. By identifying how the audio operators interact with one another, we can improve the overall performance.

4.2 Parallelisation

Many operations on audio files which are time independent such as scaling audio, highpass/lowpass can be parallelized to reduce overall running time. Parallelizing operations like FFT reduces the running time significantly. Combining this optimization with cross operator optimizations will improve the efficiency by a lot.

4.3 Linking LLVM/MLIR

We can link the Decibel compiler with LLVM/MLIR to improve the overall performance. By linking LLVM, many optimizations such as dead code elimination, function inlining can be easily supported. We can also create custom supporting passes for cross operator optimization, parallelizing audio operations in LLVM/MLIR and generate efficient code.