Introduction

The presence of electronic music is one of the defining traits of contemporary music; however, to many musicians, it may feel daunting to make the step from traditional instruments to electronic music. E-CATZ, aims to help artists bridge this gap. E-CATZ is a statically typed programming language created for musicians to easily and intuitively create music through code. With E-CATZ, artists will be able to compose music without the limitations of physical instruments, allowing far more creative freedom. E-CATZ aims to strike a balance between being intuitive for the classically trained musician and facilitating music creation that is uniquely digital. Thus it has Java-like syntax but draws reserved words from a musician's vocabulary and includes built in operations specific to music creation. It also includes a robust standard library. The library includes Har, which allows notes to be played at the same time, and Seq, which allows notes to be strung together. The standard library also includes built-in functions for randomness, an essential and unique feature of the digital art space. The language will also be designed to easily allow the creation of MIDI files, which is the industry standard for electronic musicians. Ultimately, E-CATZ provides users an accessible way to create their own musical projects, build and transform existing pieces, and experiment with sound.

Lexical Conventions

In the E-CATZ language, tokens can be separated into the following categories: identifiers, keywords, constants, pitch values, expression operators, and other separators. Whitespace and comments are ignored except in the case they are contained in a string. Whitespace is required to separate identifiers or constants that would otherwise be adjacent to each other.

Comments and whitespace

The characters /* introduce a comment, which terminates with the characters */. Anything between these symbols is ignored by the compiler; whitespace characters are also ignored.

Identifiers

An identifier is defined as the name of a variable. It must be a sequence of letters and digits; the first character must be alphabetic. The underscore "_" counts as alphabetic. Upper and lower case letters are considered unique.

Keywords

Keywords are a specific collection of characters which are not to be used as identifiers. Keywords include the name of primitive types, functions defined in the standard library, built-in expressions, rhythm, boolean, and Note values, and the words new and def. The following identifiers are reserved for use as keywords, and may not be used otherwise:

Primitive types	int, bool, float, char, Note, String
Standard library functions	read, write, randInt
Control flow	if, else if, else, for, while, break, continue, return
Rhythm values	ts, st, et, qr, hf, wh
Boolean values	true, false
Pitch values	[a-g](#|@)?[0-9]
Other	new, def, include

Pitch of a Note

All Note data types have a corresponding pitch value which can be specified in it's instantiation. Specifying pitch follows a convention which is unique to any other values in the E-CATZ language. Included in the above table are keywords for defining the pitch of a specific note that may not be used otherwise. The regular expression that defines all of these keywords is [a-g](#|@)?[0-9].

Constants

There are four types of constants or literals supported in the E-CATZ language: ints, floats, chars, and Strings. Integer constants are a sequence of digits while floats are decimal numbers. Floats must be written with periods. They can be either digits followed by a period, a period followed by a digit, or a period with digits on both sides (so .5, 1.5, and 1. are all valid floats) Characters are a single ASCII value surrounded by single quotes; Strings are a sequence of characters (ASCII surrounded by double quotes).

Expressions

The subsections are listed in their order of precedence. The convention is that higher precedence operators are performed first.

Primary Expressions

Primary expressions that involve the use of . or function invocations will group left to right.

Identifier

The type of an identifier is determined by its declaration. We define variable identifiers to be identifiers that do not precede the '()' signs.

(Expression)

The type and value of an expression are unaffected by the presence of parentheses.

Operators

Unary Operators

! expression

The result of the logical negation operator ! is true if the value of the expression is false, and false if the value of the expression is true. The type of the result is boolean. This operator is only applicable to booleans.

- expression

The result of the unary minus operator is the negative of the expression, and has the same type. The type of the expression must be char, int, float, or double.

Semitone Operators

id-expression++

The result of the expression is the value of the data type referred to by the id expression incremented by one pitch unit if id is a Note, and one integer if id is an int or float.

id-expression--

The result of the expression is the value of the data type referred to by the id expression decremented by one pitch unit if id is a Note, and one integer if id is an int or float.

Octave Operators

The octave operators << and >> group left-to-right.

expression <<

The left operand must be a Note. The result is the type of the left operand shifted down by one octave.

expression >>

The left operand must be a Note. The result is the type of the left operand shifted down by one octave.

Multiplicative Operators

Multiplicative operators *, /, and % group left-to-right.

*expression * expression*

The binary operator * signifies multiplication. Only one operand can be of type array, and the other operand must be type int. Both operands are allowed to be type int or float. If both operands are int, the result is an int. If both operands are float, the result is a float. If one operand is an array , the result is the concatenation of that operand type by however number of times specified by the int operand. No other combinations are allowed.

expression / expression

The binary / operator indicates division. Both operands must be type int or float, and the result is an int or float.

expression % expression
The binary % operator yields the remainder from the division of the first expression by the second. Both operands must be int or float, and the result is int or float.

Additive Operators
Additive operators +, - group left-to-right.

expression + expression
The result is the sum of both expressions. Both operands must be of the same type. The operand types allowed are int, float, or array. For ints and floats, the result is the sum of the two numbers. For arrays, the result is the right operand appended to the left.

expression - expression
The result is the difference of both expressions. Both operands must be type int or float.

Relational Operators

expression < expression

expression > expression

expression <= expression

expression >= expression

The operators <, >, <=, and >= all yield FALSE if the specified relation is false and TRUE if it is true. The operands must be both type int, float, or Note.

Equality Operators

expression == expression

expression != expression

The equality operators have lower precedence compared to the relational operators. The == operator returns true if both operands have the same value. The != operator returns false if otherwise. Operands must be of the same type. They can be int, float, Note, or arrays. Arrays match on contents and notes match on pitch values.

Logical Operators

All logical operators listed are left associative.

expression && expression: Returns true if both operands are true, and FALSE otherwise.

expression || expression: Returns true if one of the operands is true, and FALSE otherwise.

Assignment Operators

Assignment is right-associative and returns the assigned value.

id = expression: Assign the result of expression to the identifier corresponding to id.

Assignment is also supported in the following form:

ID += expression

ID -= expression

ID= expression*

ID /= expression

Where the left expression is an identifier, and both expressions are floats or ints. These perform the given operation on the identifier and reassign the value to the identifier.

They are equivalent to the following form:

ID = ID operation expression

where *operation* is +, -, *, or /

Statements

Expression Statement

Our statements will primarily take the following form:

expression;

Expression statements are used for assignments or function calls.

Conditional Statement

The conditional statement has the basic forms:

if *(bool_expression) { statement }*

if *(bool_expression) { statement } else {statement}*

In both cases, the expression is evaluated. If the expression evaluates to true, the first statement is executed. In the second form, if the expression evaluates to false, the second statement is executed. The dangling else ambiguity is resolved as brackets are required surrounding the statements.

The language also supports an abbreviation for nested loops in the following form:

if *(bool_expression_1) { statement_1 } else if(bool_expression_2) {statement_2} ... else {statement_final}*

In this form, the boolean expression is evaluated sequentially. The statement corresponding with the first boolean expression that evaluates to true is executed. If no boolean expression evaluates to true, the else statement is evaluated.

While Statement

The while statement has the form

while (bool_expression) { statement }

The substatement is executed repeatedly so long as the value of the expression remains true. The test takes place before each execution of the statement.

For Statement

The for statement has the form

```
for (expression1; bool_expression;  expression3) { statement }
```

The first expression initializes the loop, the second specifies the conditions that must be satisfied before each iteration, and the third indicates an action taken after each loop, typically an increment or decrement of the value initialized in the first expression. The statement is executed repeatedly so long as the value of the expression remains true.

An equivalent statement in the form of a while loop can be written as follows:

```
expression1;

while (bool_expression) {

    statement;

    expression3;

}
```

Break Statement

The statement *break;* causes the smallest enclosing while or for statement to terminate.

Continue Statement

The statement *continue*; states that we should move back to the top of the smallest enclosing while or for statement.

Return Statement

The return statement enables a function to return to its caller, and takes either of the two following forms:

return;

return expression;

The first case does not return a value. The second case returns the value of the expression to the function caller. If necessary, the expression will be converted to the type of the function in which it appears.

Include Statement

The include statement takes the form:

include "filename";

This call indicates the given file is to be copied into the current file so that its contents can be accessed. This is necessary due to the E-CATZ language's reliance on files in the standard library.

Scope Rules

The source text of the program may be kept in several files, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out through explicit calls. That is, if a function exists in one file, like a standard library, it can be directly called in any other file, without needing to be rewritten or specifically initialized.

Lexical Scope:
The scope of an identifier exists from when it is declared to the following right braces or the end of the file. That is to say, if an identifier is first declared inside a function definition, a loop, or an if/else statement, it will only exist inside that context. Identifies not declared inside braces will be global.

An identifier cannot be re-declared; this is an error. It can however be reassigned to another expression of the same type.

Declarations

There are two types of declaration supported: declaration of primitives and declaration of arrays. Our primitive types are as follows: int, double, bool, String, and Note. These can be declared explicitly in the form *datatype variablename ;* or *datatype variablename = value;*.

Note has five overloaded constructors and can take zero to three parameters. Invoking the Note constructor with no parameters creates a Note of the default pitch, c4, rhythm of a quarter note (qt), and velocity of 128. Else the constructors are are as follows:

No parameters	Note n1= new Note();
Pitch or MIDI value	Note n2= new Note (a#4); / Note n2= new Note (70);
Pitch/MIDI value and rhythm	Note n3= new Note (a#4, ts);
Pitch/MIDI value, rhythm, and velocity	Note n4= new Note (a#4, ts, 1);
Pitch/MIDI value and velocity	Note n5= new Note (a#4, 1);

Arrays are collections of zero or more items. Arrays will be declared with their type and with either a list of comma separated values for it to contain or with the size of the array. Here is how declarations will look:

```
type[] ID = [ids or literals separated by commas]

type[] ID = type[expr]
```

The first makes an array with the contents listed; the second makes an empty array of the size indicated by the expression.

Some Extra Info on Notes, Arrays, and Functions

There are three particular expressions that are important to call out.

First are Notes which are data types that contain the information of pitch, rhythm, velocity, and midi value. Declaring new notes is outlined in the section above. You can access the different members of the Note by using the dot (.) operator followed by the name of the member. To access pitch, rhythm, velocity, and midi values, you would use ID.p, ID.r, ID.v , and ID.m respectively. These values can also be edited by doing ID.*member = expr;*

Array elements can be individually accessed or sliced using brackets containing indices. The first expression below will access one element at the specified index. The second expression will access a slice starting from the first index up to the second index (non-inclusive).

```
ID[expr]

ID[expr:expr]
```

You can access the length of an array by using ID.length . Arrays can be concatenated using the + operator. By doing the expression Array * Int you can repeat the contents of an array.

Functions are callable blocks of code. By specifying the return type, identifier, and parameters, statements will be grouped such that they can be called at other points in the code. Declaration of a function looks like this:

```
def type ID(parameters) {

   statement

 }
```

Functions are called like this:

```
ID(arguments);
```

Parameters is a list of comma separated type-ID pairs such as (int x, Bool y, String z). *Arguments* is a comma separated list of expressions that the function will take in as input. Both *parameters* and *arguments* are optional and can be left blank.

Standard Library

The E-CATZ language relies on a standard library that contains the Sequence and Harmony types which are effectively special Note arrays. These data types are defined in the standard library and as such their constructors must be invoked when creating a new variable of those types. Note, while not in the standard library follows the same conventions in declaration. The format is *datatype variablename = new datatype (param(s));*.

Sequence and Harmony constructors take one argument, an integer that denotes size: Sequence s1= new Sequence(int size); or Harmony h1=new Harmony(int size);

They can also be constructed with a list of literal values, as done in a regular array, shown above.

There will also be at least three very helpful functions in the standard library: read, write, and randInt.

read will take as a parameter the name of a MIDI file and return a Note array that represents the contents of the file:

Note[] ID = read(*filename*);

write will take a note array, an int BPM (beats per minute), and a string output file name as arguments. It will write the contents of a note array out to a midi file of the given name and at the provided BPM:

write(*array*, *int*, *filename*);

randInt is a random integer generator. It will take two ints as arguments, the first representing the minimum int that can be generated, and the second representing the maximum int, both inclusive.

int ID = randInt(*int*, *int*)

The standard library may include more features as they become necessary.

Example Code

```
/*
This is an implementation of serialism, allowing one to randomly
generate masterpieces of early 20th-century music
*/

/*
return a new array that is the notes of an input array
transposed up or down by n
*/
def Note[] transpose(Note[] input, int n) {
 Note[] output = Note[input.length];
 for (int i = 0; i < input.length; i += 1) {
   output[i] = new Note(input[i].m + n, input[i].r, input[i].v);
 }
 return output;
}

/* return a new array that is the reverse of the input array */
def Note[] retrograde(Note[] input) {
 int l = input.length;
 Note[] output = Note[l];
 for (int i = 0; i < l; i += 1) {
   output[i] = new Note(input[l-1-i].m, input[l-1-i].r, input[l-1-i].v);
 }
 return output;
}

/* 'invert' the array by flipping the distances between each note */
def Note[] inverse(Note[] input) {
 Note[] output = Note[input.length];
 output[0] = input[0];
 int initialMidi = input[0].midi;
 for (int i = 1; i < input.length; i += 1) {
   int newPitch = output[i-1].m + -1 * (input[i].m - input[i-1].m);
   output[i] = new Note(newPitch, input[i].r, input[i].v);
 }
 return output;
}

Note[] finalOutput = Note[0];
Note[] input = [c3, f3, a3, b3, c4, c#4, c4, c#4];
finalOutput = finalOutput + input;

/* randomly choose a function 25 times, concatenate the results to output */
for (int i = 0; i < 25; i += 1) {
 /*
 randInt function defined in stdlib
 it returns a random integer in desired range (inclusive)
 */
 int randomInt = randInt(0, 2);
 if (randomInt == 0) {
   input = transpose(input, randInt(-12, 12));
   finalOutput = finalOutput + input;
 }
 else if (randomInt == 1) {
   input = retrograde(input);
   finalOutput = finalOutput + input;
 }
 else {
   input = inverse(input);
   finalOutput = finalOutput + input;
 }
}

/* write is a stdlib function that will create a midi file from note arrays */
write(finalOutput, 120, "serialism.midi");
```

Citation

Ritchie, Dennis M. *C Reference Manual.* 1991. https://www.bell-labs.com/usr/dmr/www/cman.pdf