# Electronic Music DIWhy
# Music Programming Languages in Theory and Practice

BY
SULLIVAN BOYD

A Thesis

Submitted to the Division of Humanities
New College of Florida
in partial fulfillment of the requirements for the degree
Bachelor of Arts
Under the sponsorship of Professor Mark Dancigers

Sarasota, Florida
April, 2019

# Acknowledgements

My thanks go out to all of my professors and mentors in my undergraduate career, at New College and elsewhere, especially those who encouraged and instructed me in computer science and music.

This thesis would not exist without its sponsor, Dr. Mark Dancigers. My conception of the topic (and later, its implementation) was inspired by not only the electronic music classes I took with him, but also our frank discussions about music programming languages, teaching such languages, and music curriculums in general. Much of the research for the topic was conducted while acting as his research assistant for creating a piece sonifying rainfall data in the Florida area.

I would also like to thank my advisor and thesis co-sponsor Dr. Maribeth Clark, as well as my thesis's computer science co-sponsor Dr. David Gillman. Their advice for thesis, academics, and life have all been instrumental to my success, especially in my final year.

Lastly, I thank the many students I met at New College for the friendships and relationships that I gained, and to everyone else who helped me graduate.

# Contents

Electronic Music DIWhy: Music Programming Languages in Theory and Practice

Sullivan Boyd

New College of Florida, 2019

## ABSTRACT

In electronic music academia, there are programming languages that can be used to generate music or to describe signal chains for sound that's rendered in realtime, as the program is being written; they can be "live coded." These programming environments, used exclusively for sound and music, belong in the category of *domain-specific programming languages*, although in the course of this thesis they are referred to using their specific domain: *music programming languages.* While they receive special attention in the course of an electronic music pedagogy in academia (they're taught in electronic music classes), they remain largely a curiosity to outsiders, both in the general public and in electronic music professions. Because they exist in the space in between music and computer science, few claim to be experts in the subject overall. Critiques on the usability of music languages are sparse; in academia, both in music and computer science, well-researched critiques on these languages are effectively absent altogether. Some of these languages are a sort of outsider art, written completely outside of academic or professional contexts altogether, while others are the culmination of academic research and/or commercial products. This thesis describes methods of comparing programming languages in general based on the features they offer, the quality of the "backend" that turns programs in these languages into sounds, and the usability of such languages for electronic musicians of average computer literacy. It then uses these methods and some basic computer science rigor to compare several specific music programming languages. Projects and examples for generating sound and music in these languages, in general-purpose languages like C++, and using professional tools like Digital Audio Workstations (*DAWs*) are also included, and are used to demonstrate these comparisons and to suggest under which circumstances some tools are more useful than others.

Mark Dancigers

Division of Humanities

# Chapter 1

# Introduction

Many fields often implement domain-specific programming languages as a form of convenience, allowing them to represent common tasks in shorthand or to automate more complicated sequences. Although popular in electronic music academia, such languages' usefulness both in the music classroom and professionally in the studio or onstage may be overall limited, and a comprehensive evaluation of their tradeoffs may not yet exist. In electronic music, such domain-specific languages can be used to automate any musical activity, from formatting scores to generating notes and synthesizing sounds in realtime for a performance. Frequently, students and teachers alike may have difficulty determining how to perform simple tasks in such languages, or even if such a task is possible in the tool to begin with. Thus, determining which tool to use or to teach can be a difficult and time-consuming process in its own right.

Programming languages, like any software, might fall prey to countless mishaps, especially if they evolve organically over time, becoming a sprawling mess or otherwise too bulky to use. The standards (and sometimes the implementation) of more popular general-purpose languages are frequently determined by large committees taking input from users both amateur and professional. If the project is maintained and used by a small enough demographic (i.e., only a few people), solid critiques on the language that would keep it on track in its development path as a useful tool might be conspicuously absent. Because music is an art and not directly inside of or attached to the computer sciences, domain-specific music programming languages (*music programming languages* hereafter) exist as something of an outsider art in both the musical and computer science realm: they are often written by amateur programmers and used by an audience that is close to completely non-technical in proficiency. As a result, while they may often have strong aspects that justify their use, elements of the language specification or in the overall workflow they belong in may consistently cause problems for the programmers using them, either because the element is genuinely useless or poorly-made, or because it requires a technical or programming prowess beyond the level of the demographic using it; the average musician is not an accomplished programmer or student of language design.

In addition, the use of such languages even in their field is somewhat niche, as commercial Digital Audio Workstations (*DAWs*) exist in the form of graphical user interfaces that boast a significantly lower bar to entry. They make use of the graphical elements to tersely represent features and information still difficult to access in a music programming language. Besides competing with the languages on their own, the DAWs usually act as a host for other musical programs by synchronizing them in time, saving and restoring state of all of the components loaded, and providing an interface for plugging the various audio components together. To be useful in a professional context, having a language attached to such software would be arguably necessary, but instead of integrating into a DAW like any other element in electronic music, programming languages frequently attempt to act as the top of the hierarchy and emulate a complete workstation themselves, preventing them from playing nicely with the remaining tools of the musical trade.

But simply integrating into a workstation like any single synth or sound effect may not solve their workflow problems altogether. In generating electronic music, music languages are frequently useful not as a plugin but as an agent that exists *between* the other plugins, able to manipulate their parameters and the connections between them. To succeed, then, a music programming language might be better-suited not as a plugin but built into the DAW itself, capable of using the script to automate any arbitrary component therein. Even then, a domain-specific language may be redundant: similar tools in other domains, like 3D modeling, frequently employ embedded general-purpose scripting languages like Python or Lua to let a skilled user generate or modify the solids they design.[1] Instead of writing an entirely new language around the tool, it is sufficient in these cases to simply extend the existing language, or to provide the extensions in the form of a library or API.

In the course of researching for this thesis, I evaluated three popular music programming languages by their feature set and by simply writing programs with them. In an attempt to tease out some of the confusion experienced by my peers and musical professionals, I also approached them from a programming language design perspective. I evaluated their syntax and grammar and noted unusual and unexplained elements of both the languages and the features of the back ends that they drive and attempted to categorize the resulting strengths and weaknesses as either mechanical or musical. I also describe briefly the components of a programming language both as software and as an abstract collection of characters so that other musicians may be able to better-evaluate their own tools when needed. A truly comprehensive collection of music programming language reviews would be meandering and inevitably inconsistent; some of the languages are used simply for formatting traditional non-electronic notation, while others deal exclusively in note generation and synthesis. Still others

1. Autodesk, "Python in Maya," 2018, accessed April 1, 2019, `https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2018/ENU/Maya-Scripting/files/GUID-C0F27A50-3DD6-454C-A4D1-9E3C44B3C990-htm.html`.

exhibit a hybrid of the two. For consistency, the languages discussed here are all in the second class: they are used for synthesizing electronic music and for generating the tones to be synthesized. I also discuss the somewhat awkward aspect of how and when these languages interact with DAWs. Some of them do exhibit the ability to be hosted in a workstation (and rarely may even be able to act as "glue" for the entire workstation as opposed to simply acting as a plugin), but in all cases this comes a the cost of other design tradeoffs specific to the family of languages implemented in these DAWs. Finally, I speculate on programming for music in practice, using domain-specific and general-purpose programming languages in and outside the DAW, both in the form of programming projects in a production context and as discussion of where the field may find itself in the future. Parallel to this examination and of similar importance is a collection of my own musical work, created in the languages discussed, or generated and performed with general-purpose programming tools and audio libraries. Through this compositional work and combined with the language discussion, common electronic music practice in various toolsets can be evaluated and informed decisions can be divined from the benefits, costs, and risks of each.

# Chapter 2

# Programming Languages

## 2.1 Definitions and Types of Programming Languages

Judging the qualities of a music programming language requires to some extent the ability to form opinions on the grammar and syntax of the language itself; this section serves as an introduction to common terms used in programming language design and in programming in general.

### 2.1.1 Imperative Languages

Programming languages for describing a process or an algorithm are usually **imperative**;[1] their syntax and purpose are built around executing a series of commands, usually mathematical in nature, and they're easily the most prolific. A defining feature of imperative languages is the use of an algebraic grammar in the form of

1. Aho et al, *Compilers: Principles, Techniques, and Tools*, 2nd ed. (Addison Wesley, 2006), ISBN: 0321486811.

*expressions*, e.g.

```
x = 2.3f - 5.3f;
```
**Figure 2.1:** An imperative line of C code that assigns a number to *x*.

in the programming language C and others derived from it. The expressions are built around *literals and constants* (like numbers), named *variables* (the `x` above), and *operators* like the `=` and `-` characters in the example above. Operators usually represent an instruction or an operation that acts on *operands*, the constants and variables to the left or to the right of the operator. Operators have an *arity*, which is the number of operands that bind to it. In common use, most operators are only unary or binary, meaning they accept only one or two operands. Mathematical operators like `+` and `-` are binary, while for example (in C and others) the logical inversion/NOT operator (e.g., `!x`) only takes one input and returns the inversion (true or false, whichever was not already the case). Another common unary operator is the increment operator, `x++`, which e.g. adds 1 to `x`.

Some languages allow their operators to be *overloaded*, which changes the operation performed by the operator depending on the context, usually the type of the operand(s) it has taken. For example, a `+` operator in most languages will add two scalar numbers, like `2` and `5` together, i.e. `2 + 5`. A programmer might provide an overload for the same operator that functions instead on vector numbers, like `(2,3) + (5,6)` in an environment where such a vector type exists. Overloads can be specified by the language itself (following the `+` example, most languages provide an overload that performs addition on each numeric type), and many modern languages (like C++, Python, and Rust) allow programmers to specify their own overloads of the existing operators. While some languages like Java do not allow the programmer to overload their own operators, essentially every typed language needs to provide overloaded operators in their core specification for all of the relevant contexts the operator may be used in (eg, the addition operator in Java can be used on both numbers and to concatenate strings of letters, like `"hello " + "world"`). While overloading is usually necessary for a language, excessively overloading an operator makes the purpose of a statement using the operator unclear both to a reader and potentially to the compiler.

Both language developers and programmers alike are somewhat split on the exact etiquette of overloading, with arguments existing both for it being limited in common practice and for restricting its usage by programmers altogether. As a result, some languages like Java simply don't allow user overloading of operators under any circumstances. Java is based in many ways off of C++, and contains differences in the forms of grammatical changes and other tradeoffs that were developed mostly in response to perceived failings of C++. The decision not to allow programmer-defined operator overloading in Java has been somewhat divisive[2] (although many are mostly neutral to the matter), with programmers dealing with math and geometry often particularly in favor of defining their own operator overloads, and a somewhat smaller but more vocal group decrying the feature after having experienced overloads written by other programmers that make code difficult to read and to modify. In all of these schools of programming, it is generally agreed that some care should be taken that any overloading performed is necessary and makes sense to anyone reading it. Language specification-defined operator overloads usually cover only the bare minimum swatch of contexts required to program in the language; any overloads that might be outside of this swatch fall under scrutiny.

The CPU inside of a computer runs its programs by executing a simple sequence of the commands hard-wired onto the chip; imperative programs, which represent a list of commands to be run in a sequence, map well to programs that need to describe

2. Herb Sutter, "The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling," *Java Report* 5, no. 7 (2000).

complicated tasks and algorithms for the CPU to complete. To convert the high-level programming language code to the low-level machine code the CPU can read, the code must be *compiled* or *interpreted*. Compiling is usually done offline by a compiler, which saves the resulting machine code inside an executable file so that it can be run later. An interpreter on the other hand reads source code and then executes it on the CPU immediately at run-time. To facilitate this process, the compiler or interpreter may instead map the source code to an *intermediate representation* (IR) that is lower-level than the source code but still higher-level than the machine code. Such an environment might also employ a *virtual machine*, which runs on the CPU but itself emulates a higher-level virtual CPU that can read and execute the IR.

Modern CPUs actually have multiple smaller units inside, often referred to as cores. Each core is capable of running its own program or sequence of commands independently of the others, allowing for multiple programs to run at a time, or for a single program to do multiple things at once. With or without these multiple cores, the operating system scheduling the programs usually wants a way both to multitask simultaneously-running programs and to provide the programs with an interface to run multiple tasks concurrently themselves. The programs can create *threads* of execution that are usually each assigned to their own core, resources allowing. Creating a thread is called *detaching* or *launching*, while waiting on another thread to finish before continuing is called *joining*. To manage many threads with limited resources, the operating system or virtual machine implementing multitasking needs a way to switch from one context to another. There are two such schools of multitasking. The first involves the scheduler automatically interrupting a running thread periodically to allow a suspended thread to resume its work: this is preemptive multitasking, or *time-shared scheduling*. The second requires programs to manually indicate that they are ready to suspend work and to *yield* to another process: this is non-preemptive multitasking, or *cooperative multitasking*. A programming language environment might implement its own form of scheduling (for example, in its accompanying virtual machine), or it might provide support for accessing the operating system's scheduler to control tasks at a lower level. In a threaded or time-shared scheduled environment, the language might provide functions for launching and joining threads. Conversely, in a non-preemptive environment, the language would usually have a `yield` keyword or an equivalent to manually save the state of the process and return the execution to the scheduler. When the other processes have also reached a yield in their code, execution on the original process can be resumed on the line after the yield.

The binary digits representing a variable in the computer's memory might have multiple meanings depending on the context. Computers are only capable of counting using ones and zeroes and have no built-in concept of letters, decimal points, colors, etc.[3] Modern CPUs do have instructions for processing a variable in different ways, but the digits themselves remain ambiguous and the instruction to be used needs to be specified in the program. The binary number `10101010` might represent the unsigned (positive) integer `170`, the signed integer `-86`, the lowercase `a`, or a fractional number with a decimal point. Most CPUs will have unique instructions to deal with each type; a program running on the CPU needs to specify in advance which instruction is going to be used. Some low-level programs where performance is important can take advantage of this property and treat one set of bits as one type and then later as another in order to achieve some performance advantage (usually at the cost of some confusion in the code). A programmer using a high-level imperative language should be able to tell the CPU which type of instruction to perform on a variable once when the variable is declared to prevent accidentally performing undesired instructions: this is called *typing*, and is present in some form in almost every modern language.

3. Nisan and Schocken, *The Elements of Computer Systems: Building a Modern Computer from First Principles*, 1st ed. (MIT Press, 2008), ISBN: 9780262640688.

```
int myInt = 0;
float myFloat = 0.0f;
myFloat = (float)myInt;
```

**Figure 2.2:** Declarations using types in C and explicit casts between the types.

The *strength* of a type system is how tightly it requires a programmer to use its type system. A more weakly-typed language allows for its variables to be *implicitly cast* from one type to another with relative ease; a more strongly-typed language requires more verbose *explicit casts* from one type to another, or in some cases may not allow casting at all. Some languages (like C, in Figure 2.2) require the type being used to be specified in the declaration. Others, like Python or Lua, will automatically detect the type of the variable when first defined.

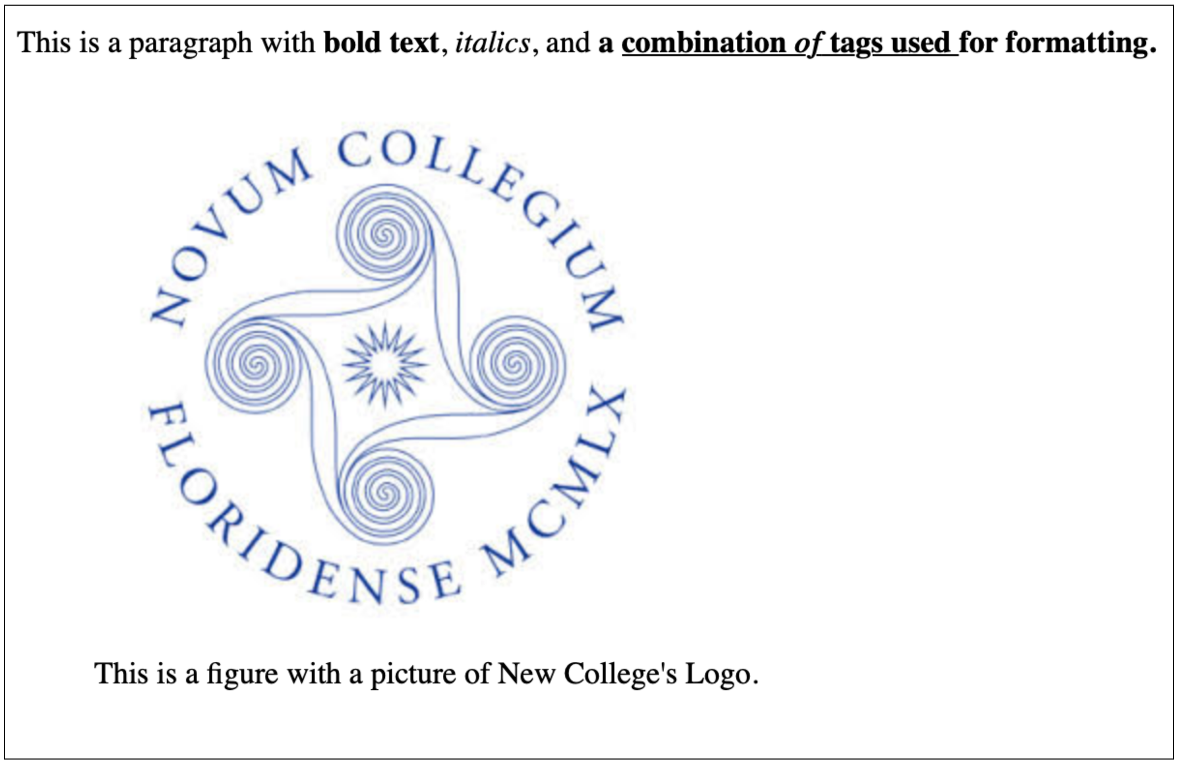
### 2.1.2 Declarative Languages

Languages used for generating a document or some other static output are generally called **declarative** or **markup**; their syntax and justification is built around things like formatting in a word processor, or wirings between synths and effects in a signal chain. Unlike imperative languages, where a command in the code often corresponds more or less directly with a similar command being executed on the CPU, a declarative language is essentially a series of decorations that wraps around unformatted content. One popular declarative scripting language is HTML, or HyperText Markup Language. HTML describes a web page by using bracketed tags to delineate both larger sections of the page, media and special content like hyperlinks and images, and formatting on individual characters of the text, as shown in the figure below:

```
<body>
<p>This is a paragraph with <em>bold text</em>,
<i>italics</i>, and
<em>a <u>combination <i>of</i> tags
used </u> for formatting.</em>
</p>

<figure>
<img src="images/fourwinds.png" alt="New College’s Logo"/>
<figcaption>This is a figure with a picture
of New College’s Logo.</figcaption>
</figure>
</body>
```
**Figure 2.3:** Declarative HTML describing the body of a web page.

When rendered in a web browser, the HTML displays the results similar to Figure 2.4.



This is a paragraph with **bold text**, *italics*, and **a <u>combination *of* tags used </u>for formatting.**

This is a figure with a picture of New College's Logo.

**Figure 2.4:** The page body rendered from the HTML

## 2.2 Music Programming Languages

Aside from the general purpose programming languages used as examples above, there exist languages designed and implemented specifically for use in a unique field. These *Domain Specific Langauges* can reflect their domain's needs through both specializations in the language's syntax and grammar and in the runtime environment, which may provide special libraries or patch the interpreter into a special model or simulation used by the domain. One such domain of languages may compile to a sound instead of a series of raw CPU instructions or a formatted document. Such languages might work offline and fill a file on disk with the produced sounds, or they might use a virtual machine to interpret the written code and render it directly to a sound card in realtime. Some offer deep per-sample control over the output waveform, while others are more high-level and are better at editing or sometimes only declaring a signal chain and then tweaking parameters on the signal generators and effects. Music programming languages are usually interpreted in realtime to allow for "live coding," and frequently use a virtual machine to evaluate and re-evaluate the source code to produce a continuous output signal for as long as the program is left running. They frequently use a mix of imperative commands to describe processes but also declarative sections that specify note sequences, rhythms, signal chains, and pitches.

Rendering audio is an expensive process for the CPU. When issuing commands to an audio stream, music programming languages have to make optimizations to enable them to run in realtime, usually in the form of tradeoffs to the audio quality. For example, a language environment might opt to implement a less-accurate synthesis algorithm in favor of a simpler one that's less taxing on the CPU. Another option is to provide multiple implementations of a synthesis or analysis algorithm; the responsibility of creating a lightweight program then falls to the programmer, who must choose the implementation to be used and accept the corresponding trade-offs. Along the same lines, many of the computations the programmer expects to be performed (even outside of synthesis) in an update loop can be simply too taxing to be performed all together in a single audio frame. To compensate for this, many language environments implement update loops at two different intervals: one at the *sample rate*, or SR, which is run for every single sample rendered, and another at the *control rate*, or CR (frequently KR), which is run much less often, usually only a few times a frame, or at an interval specified by the programmer.[4] Programs executed at the control rate are usually fast enough to give a sense of "animation" (useful for automation of eg volume or other instrument parameters) but are still much computationally cheaper, while sample-rate execution is comprehensive enough for the entire audio synthesis process itself but can potentially overwhelm the CPU.

Common among music programming languages is that they are (at least in origin) somewhat orphaned academically speaking. Because they exist in a shared space between music and computer science, and because the projects are often maintained by small teams (or as few as a single person), their design and their functionality is often focused in the domain the creator has more experience over. For project developers with greater strength in the computer science domain, music programming languages often manifest as further specifications or additions to existing popular general-purpose programming languages (or, more frequently, as simply a library and API), like *Overtone*, a music programming library built for the Clojure language.[5] The languages discussed here are built in some ways with a stronger background in music than in computer science. They are all original domain-specific languages designed specifically for the task rather than extensions of existing languages. Their

4. Andrés Cabrera, "An Overview of Csound Variable Types," *CSOUND JOURNAL* 1, no. 10 (2009).

5. Hloover Sigurosson, "Overtone User's Guide," 2018, `https://overtone.github.io/docs.html`.

musical focus often leaves their syntax and grammatical choices mostly unexplained in their documentation and specifications, while their use in electronic music is outside the traditional workflow in an audio workstation (although they've gained significant popularity since some now have the option of operating inside of the DAW's workflow). In this way, through their split focus, few (and often amateur) personnel, and (until recently) niche audience, discussions regarding the quirks and actual raw usefulness of the language as a tool are frequently pushed to the side or excused as a side-effect of having few accomplished programmers in the field who can mentor students. Music programming languages like the ones discussed below, then, are **outsider art**: they stand on the periphery of academic and popular knowledge in the field. Their following (or flouting) of conventions both computational and musical are occasionally intentional but are just as often completely arbitrary and unjustified by their creators.

### 2.2.1 ChucK

ChucK is a primarily imperative music programming language among whose primary features is its concurrent programming model that provides syntax for both more concise and more precise control over time and rhythm than other languages.[6] The syntax is also distinct: its mostly C-based style is peppered heavily with its own unique (and eponymous) ChucK operator, `=>`, which by its own admission is "massively overloaded" to the point that the thesis describing the language includes only an abbreviated list of all of its overloads and simply ends the list with an enthusiastic "...and more."[7]

The ChucK operator's primary and arguably most intuitive purpose is not imperative

6. Ge Wang, "ChucK : Strongly-timed, Concurrent, and On-the-fly Music Programming Language," 2015, accessed July 8, 2018, `http://chuck.cs.princeton.edu`.

7. Ge Wang, "The ChucK Audio Programming Language: An Strongly-timed and On-the-fly Environ/mentality" (PhD diss., Princeton University, 2008).

but declarative: it represents a static connection between elements in a signal chain. It is also used in imperative contexts for other music-specific uses, like advancing time. But the operator can also be used as syntactic sugar for function invocations. Furthermore, it's also used extensively for assignment, replacing the `=` operator used in its place in nearly every other language in existence. The assignment operator used in most languages follows the form used in Figure 2.1, with the destination variable on the left and the content to be assigned on the right of the operator. ChucK's version using its overloaded operator eschews convention and places the destination on the right instead. With assignment comes compound assignment, the process of performing both arithmetic and assignment in one line of code (usually formatted in most languages as `x += 2;`), so the `=>` operator also has a whole host of slight variations in the vein of `+=>`, `-=>`, and so on.

In addition to providing necessary functions specific to the music domain, the ChucK operator also tries to take over those already commonplace in practice in most other programming languages. In ChucK, `=>` effectively does everything. While in another programming language, function calls, assignments, and concurrency yields have vastly different syntaxes that form shapes on the page that are more or less immediately differentiable from each other, a programmer reading ChucK must take a much slower look at their code in order to determine the context the single `=>` is being used in to begin to understand the meaning of the line. Even when writing and editing ChucK code, the programmer must be careful not to mistakenly add (as one example out of many) an assignment to a signal chain, both of which are virtually indistinguishable from each other visually on the page. Such an error would only become known at runtime when the environment spits out an error, as there's nothing making the syntactically incorrect code particularly visually different from a valid statement. The heap of overlapping meanings was done with some intention and is introduced in ChucK's literature with some fanfare, but its purpose (besides to lead programmers astray) is never laid bare across any of the creator's many articles on the language, academic or otherwise.

```
// Connections in a signal chain
noise => filter => dac;

/* Function invocation
(C-style, still valid in ChucK)*/
Math.min(a, b);
// Function invocation with overloaded operator
( a, b ) => Math.min;

// Assignment
2 => x;
// Compound assignment
2 +=> x;

// Time advancement:
// Yield to the scheduler and come back in 2 seconds
2::second => now;
```

**Figure 2.5:** Examples of overloads on the ChucK operator

The ChucK language is cooperatively multitasked, being composed of multiple files running concurrently that it calls *shreds*. Shreds yield to the virtual machine as a form of time control, which is reflected in the syntax of the yield statement (once again using an overloaded ChucK operator), as shown in the final example in Figure 2.5. The listed example suspends the shred for two seconds and then resumes execution. As an example: to play a series of notes in a rhythm, the program could set the pitch of a synth, yield for some time, and then set the pitch again for each note in the sequence. Executing the code to set up a signal chain or to set the pitch of a synth takes only a minute amount of time. When the virtual machine finishes running code in shreds and has such time to idle, it begins sending rendered sound to its specified output using the settings and logic specified by the most recently-executed ChucK code. In the note series example, it renders the output of the synth that the most recently-run shreds had set to a certain pitch. ChucK's code, then, works primarily at the Control Rate, as it only updates at user-specified intervals rather than at every sample. However, per-sample synthesis can still be specified with the same code, using `1::sample => now` to increment by a single sample instead of by a couple of seconds.

ChucK's virtual machine also performs another important and somewhat unique function: it can accept shreds from other computers on a local area network connection, as well as commands from these computers to start, stop, and replace active shreds already on the VM. This allows multiple programmers to send code to the VM, each representing some kind of time-sensitive musical material, and have it all output synchronously (actually, the code is executed asynchronously as above, but the mixed audio output generated while the shreds are waiting uses all of the active shreds together). In other words, performers can write code onstage and have their various programs compiled on the fly and sounding together in realtime, all while keeping the rhythms synchronized between shreds: this is *live coding*, and ChucK is unusual in allowing such collaboration over a network.

### 2.2.2 Max and Other Patching Languages

Distinct from text-based programming languages are the family of graphical programming languages that began with *The Patcher* in 1988[8] and continue to be supported to this day in the form of the commercial software, *Max/MSP* and the open *Pure-Data*, as well as in various spin-offs, imitators, and similar products, like Native Instrument's *Reaktor*. These languages are more or less patch-based in that they resemble the patches of analog synths made with wires between the modules as a way of wiring ("patching") a signal from one processor to another. Historically only used for message passing, *The Patcher* initially could thus only output control messages to

8. Miller Puckette, "The Patcher," *Proceedings, ICMC. San Francisco: International Computer Music Association*, 1988, 420–429.

external synths. Modern versions also have the ability to perform signal processing and sample-rate computations, and most projects in modern Max usually output an audio signal in realtime to the sound card. Despite being laid out graphically, the languages are similar to ChucK and SuperCollider in that they consist of mostly pre-configured unit generators and effects that can be patched together. Max specifically also has a somewhat unique "feature" as a result of being commercial software owned by a larger company: it alone is available as an integration into an audio workstation (but again as a result of its status as intellectual property, it is locked to a specific workstation, Ableton's *Live*). Max's *Live* integration allows it to create modules that act as synths or audio effects, but also for those modules to modify the parameters of other synths and effects. In other words, it can be used both as a tool for generating and modifying audio or for writing scripts that automate the behavior of other components used in the DAW. But Max's ability to enact this type of automation is somewhat at odds with the language itself.

Syntax in all of these languages is similar: a series of rectangles (*objects*) with words patched together with "wires" that creates a directed graph that information and events flow through. Each type of object can have a number of *inlets* for information to flow in and some *outlets* for any outputs. The rectangles representing the objects on the graph have some form of interface inside them; some objects simply contain a name and can only be controlled with outside inputs, while others may have a complete user interface inside their rectangle. This is done with intention: text-based objects using inlets and outlets are the default mode of representation, while graphical representations are primarily reserved for editing or displaying groups of data in eg list or dict format. The languages are more or less weakly-typed: integers, floating point numbers, and audio signals are common types used in the Max family; lists and arrays, dictionaries, and a "bang" type for triggering updates are also supported. Most of the types are sent between objects as *messages*, meaning they flow through the graph at the control rate only a few times a second. Signals are distinct in that they necessarily flow at the sample rate. Conversions between types are somewhat inconsistent; implicit conversions of a typed wire can occur at the inlet to an object and are highly contextual. The results (and whether or not such a conversion is even allowed) depend upon the objects and types in question. One way the language environment prevents unwanted implicit conversions is by offering two versions of some objects: the first operates at the control rate and thus outputs eg. float or int messages, while the other operates at the sample rate and usually outputs a continuous signal. The sample-rate version is distinguished with a tilde ~ at the end of its name but is otherwise usually identical. Most of the maintained languages in the family (Max and PureData at least) can be extended by programmers with new objects; an API is provided in C (with bindings for other languages) for creating and defining new objects.

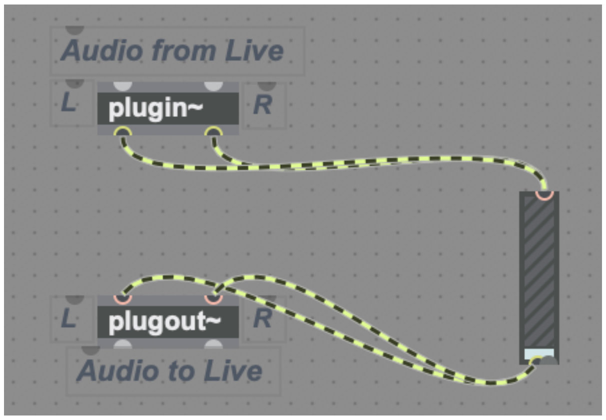


**Figure 2.6:** Max patch integrated in Ableton's *Live* DAW

Many common objects in Max and its friends are *functional*: that is, they take some inputs and produce some outputs as some function of the inputs received without changing the state of any of their inputs. For example, arithmetic operations that would use an operator in an expression in an imperative language take up a single box/object in Max. Performing a simple addition of two numbers for instance uses a + object with the inputs piped in through the inlets above. A pair of audio signals can be added in the same way at the sample rate using `+~`. This functional approach lends itself well to audio processing, as the flowgraph can represent a series of computations performed on every sample without the need for additional control logic like a loop that would be necessary in an imperative language.

But other objects can forego this format and not have any proper inlets or outlets at all, instead using (for example) named variables in an invisible global namespace to communicate with other objects. Still other objects have little to do with a functional design paradigm at all, like `if`, an object with a decidedly imperative flavor that nonetheless plugs into the rest of the flowgraph as usual.[9] The `if` object takes up to nine inlets and has two outlets. The text inside the object is written much like an if statement in Lua, with an `if [expr] then [statement] else [statement]` form, and operates at the control rate (that is, there is no `~` in its object name). Performing math in the rest of the language is functional, using a flowgraph of operator objects, but `if` expects its test expression written in an imperative style. Likewise, the outputs, while still flowing from an outlet back into the rest of the flowgraph, must be changed imperatively with `set [output]` statements (technically, each token following the `if` (`else`, `set`, etc) is treated as a parameter of the `if` object by Max, but the complete statement must be parsed internally in order to function). The `if` object reveals a tension at the core of the language's intended programming experience: some of it works well with the graphical functional style, but other elements still strongly suggest an imperative method of programming, even though an `if-then` itself can technically be represented as arithmetic on a flowgraph. That `if` is a control-rate object provides some hints as to when it's useful: the functional style is more popular for audio-rate processing, but the language's strengths falter when performing control-rate adjustments (such as automating parameters or generating

9. Cycling 74, "if Reference," accessed March 18, 2019, `https://docs.cycling74.com/max5/refpages/max-ref/if.html`.

harmonies) that require or at least benefit from imperative statements like the ones `if` uses.

Since the differences in programming style are mostly hidden (even `if` still plugs into the flowgraph), a musician without programming knowledge might fail to understand which paradigm works better in what roles. The mingling of control- and sample-rate objects, while useful in its own right, can throw further confusion into the mix: a novice musician-programmer may attempt to use an `if` in their audio processing, only to discover later that their use of the imperative object has implicitly converted their sample-rate signal to a control-rate variable (and thus that their program is no longer producing any audio at all)! An attempt at creating an optimal work environment for audio programming might try to separate these two programming paradigms and, to prevent confusion, to constrain each to its domain of usefulness. But to Max and its lineage of language, the distinctions are unimportant to their creator: "Computer science has never found a metric for determining whether or not a computer program is fun to use."[10]

### 2.2.3 SuperCollider

SuperCollider is another music programming environment designed for building sounds out of a series of unit generators and effects plugged together. Originally built as an alternative to the aging *CSound* (itself a monster of a language that somewhat confusingly represents not just instructions but synthesis and effects chaining with assembly-style opcodes),[11] SuperCollider was later adapted into a Max patch and finally into its own environment.[12] Syntactically, the language boasts similarities to an

10. Miller Puckette, "Max at Seventeen," *Computer Music Journal* 5, no. 26 (2002): 31–43.
11. "Opcodes Overview," 2019, `https://csound.com/docs/manual/PartOpcodesOverview.html`.
12. Wilson et al, *The SuperCollider Book* (MIT Press, 2011), ISBN: 9780262232692.

extensive list of general-purpose inspirations, from Ruby to C to Lisp to Javascript.[13] Certainly, the code itself is not recognizable as an obvious descendent from any single general purpose programming language, nor would an experienced general purpose programmer find much if any of their computer science knowledge at all relevant in SuperCollider's domain.

```
 Server.default.boot;

(
SynthDef(\SynthTitle, {
    var snd, freq, bw, delay, decay;
    freq = \freq.kr(440);
    freq = freq * Env([-5, 6, 0], [0.1, 1.7], [\lin, -4]).kr.midiratio;
    bw = 1.035;
    snd = { Saw.ar(freq * 0.5 * ExpRand(bw, 1 / bw)), 0.01, Rand(0, 0.01)) };
    snd = (Splay.ar(snd) * 3).atan;
    snd = snd * Env.asr(0.01, 1.0, 1.0).kr(0, \gate.kr(1));
    snd = FreeVerb2.ar(snd[0], snd[1], 0.3, 0.9);
    snd = snd * Env.asr(0, 1.0, 4, 6).kr(2, \gate.kr(1));
    Out.ar(\out.kr(0), snd * \amp.kr(0.1));
}).add;
)

(
var durations;
durations = [1, 1, 1, 1, 3/4, 1/4, 1/2, 3/4, 1/4, 1/2];
Ppar([
    Pbind(*[
        instrument: \SynthTitle,
        amp: -20.dbamp,
        midinote: 62,
        dur: durations.sum * 2,
        sustain: 7,
    ])
]).play(TempoClock(210 / 60));
)
```

**Figure 2.7:** An example of SuperCollider code

The audio simulation provided by the environment is extensive. A server renders the audio itself and receives messages to allocate, play, and automate. Busses, buffers,

13. James McCartney, "SuperCollider," 2019, https://supercollider.github.io.

frames, signal generators, and effects (the last two both called *UGens* in the language specification) are all available for manipulation by the programmer. In this sense, the environment somewhat less resembles a more traditional programming language and moreso an API provided to access the mountain of assorted features the server provides. Somewhat like Max and its language family, extensions to SuperCollider in the form of custom UGens can be made through a provided API in C. But while the environment itself was built from scratch around these resources, the syntax and organization of the language specificallly may have taken a backseat. Certainly, in recounts of the history of the language, the justifications for the grammar as a whole are conspicuously absent. Syntactic sugar abounds: messages sent to the server, for example, might take the form of `message(args...)` or `object.message`, where `object` is a specific argument.

The unusual and largely unjustified grammar in SuperCollider stands in some contrast to the quality of the provided audio server it interfaces with, which is regarded by some to have higher-quality synthesis algorithms than that provided by other audio programming environments.[14] For example, several language projects have been developed using SuperCollider's audio server as a base, thus implicitly being built around replacing the language that was built for the purpose while maintaining the superior synthesis capabilities. Notably, a front-end for the server exists for the Clojure language as a library called *Overtone*. The education-focused language *Sonic Pi* also uses SuperCollider's audio server as a backend.[15]

14. SuperCollider Users Mailing List, "SC vs Pure Data for teaching," accessed July 8, 2018, `mailto://sc%C2%ADusers@lists.bham.ac.uk`.

15. Sam Aaron, "Sonic Pi License," 2016, `https://github.com/samaaron/sonic-pi/blob/master/LICENSE.md`.

## 2.3 Other Languages

Besides ChucK, Max, and SuperCollider, there exist a few other music programming languages in common use. Perhaps the most popular of the remainder is CSound, which despite its relative age still enjoys a dedicated userbase. CSound programs are divided into several sections, including an XML (declarative) section for the *Orchestra* and *Score*, as well as a more imperative section for controlling the resulting orchestra and generating or modifying its score. Despite the name, CSound is not particularly C-like; its imperative sections are best described as an imitation of assembly code, with lines taking the format `output opcode input1, input2, ...`. While the language itself does not support or integrate with DAWs in any way, an external tool exists that can package a CSound program as a VST instrument or effect, which can then be loaded as a plugin.[16]

In addition to live performance and general music production, video games are another domain that makes use of programming for music. Usually, music and sound features are coded directly into the game's engine (or added on using a commercial library), and artists simply interact through a provided editor, but some exceptions exist. Notably in the industry, Valve Software has a language used in its more recent games to allow their composer to control playback and parameters thereon of various sounds.[17] Embedded in their *Soundscripts* are *Sound Operator Stacks* that can run when a sound event is started, ended, or every time the sound is updated. The syntax is a hacked form of the declarative Soundscripts, making reading and writing the code somewhat messy, but as a result, musicians have access to most sound engine features without having to modify and recompile any of the game's core code; the designer can control (for example) pitch, spatialization, ducking, and start or stop additional

16. "Introduction — Cabbage Audio," 2018, accessed April 8, 2019, `http://cabbageaudio.com/docs/introduction/`.
17. Valve Software, "Soundscripts - Sound Operator Stacks," 2011, accessed April 4, 2019, `https://developer.valvesoftware.com/wiki/Soundscripts#Operator_stacks`.

sounds.

# Chapter 3

# Applications of Music Programming Languages

To better evaluate the strengths and weaknesses of each music programming language, a sample program was necessary to test both their musical and technical capabilities. Ideally, such a program would be used as part of a larger composition to determine their usefulness in a production environment, where less obvious pros and cons to each language, such as basic synchronization with other technologies and overall ease of use, might manifest. I wrote such a program to sonify rainfall data for a composition that juxtaposed the procedurally-generated notation with improvisation from live musicians in a small ensemble and used it as a testbed for several music programming workflows. Because the music was performed only once to a live audience, only one of the prototypes would actually end up being used. As a result, some aspects of how the remaining workflows might perform in such an environment were unexplored. However, these remaining languages generally underperformed in the earlier stages of their exploration, hence their candidacy for the performance being discarded in the first place.

## 3.1 An Introduction to Sonification

*Sonification* is the audio counterpart to visualization. Much like a table or graphic, a good sonification provides a presentation of data in a form intuitively interpretable to the listener, while often both holding and abstracting information at a greater depth. Obviously, such a sonification should reliably represent the data sonified and present the message the sonifier wants to use without obscuring or falsifying the data's original content. Ideally, it should also be aesthetically pleasing and interesting to the end listener. I created a tool for sonifying data for use in an audio workstation environment, primarily for composers and sound designers to audition sounds in conjunction with their rules for mapping the data even as those rules are still being changed. In the process, I experimented with various workflows for authoring audio and musical programs that can generate and perform such sonifications in realtime. Critically, the sonifier/musician is able to tweak the parameters while the program is being performed in order to both better choose the algorithm that fits their needs, and to better understand the data set being used. I used the sonification project as an opportunity to compare and contrast some music programming languages with each other and with alternatives like sound APIs for general-purpose programming languages. Some solutions in music programming environments like Supercollider performed well, but the bulk of the more promising work was written in C++ and integrated into a DAW.

### 3.1.1 Types of Sonification

While visualizations and what they can represent are fairly standardized (e.g., charts, graphs, physical models) and have readily-available tools to create them, the methodologies and tools used to sonify data aren't nearly as obvious and can frequently be much more abstract. Most sonification fits into at least one of three major categories.[1] The first and most direct form of sonification converts data points directly into samples, making (sometimes) a coherent sound when played back at audio rate. This can be useful in cases when large quantities of data have unusual harmonic content (or a distinct lack thereof) that make a recognizable sound on their own. Usually, non-audio data converted to an audio format in this fashion does not reliably create interesting sounds, or any sound that is intuitively useful to the listener. Second, to provide more control over the aural content of the sonification and to make the resulting sounds yield a desirable interpretation, the data can be mapped to sounds (eg, each data point might emit a single note). Finally (and even less directly), it might be mapped to parameters of a musical model (eg, a range of data might control a direct parameter like tempo or less-directly effect ”mood” in a composition). The latter two approaches frequently lose accuracy and are subject to the sonifier's bias, as they choose the manner in which the data controls the sound parameters, but the musicality is more popular for composers.[2]

## 3.2 Project: —When it Rains, it Rains—

I was presented with climate data (specifically, rainfall and temperature) from Florida State University in comma-separated value (CSV) file format[3] and asked to create a program that could generate sounds from the input data in realtime to be performed as music in a concert form and later extended for more general sonification projects. As a compositional tool, the program needed to give the composer as much control

1. University of California, Berkeley, “Solar Wind/Education and Public Outreach for STEREO/IMPACT and Wind,” accessed July 11, 2018, `http://cse.ssl.berkeley.edu/stereo_solarwind/sounds_programs.html`.
2. James Saunders, “No Mapping”,” *MusikTexte*, no. 149 (2016).
3. Florida State University, “Downloadable Data - Florida Climate Center Office of the State Climatologist,” accessed October 21, 2018, `http://climatecenter.fsu.edu/climate-data-access-tools/downloadable-data`.

over the resulting sounds as possible without overwhelming them. Meanwhile, a generative backend was needed to plug into a traditional audio workflow, or at least provide support for mixing and mastering. Essentially, two parts were needed: the user interface that the composer would use, and the guts of the program that would automate the sonification process.

### 3.2.1 Prototyping the Sonification Tool

I began work on the sonification plugin based on a prototype of Dr. Mark Danciger's that was built specifically for the piece he was composing, *When it Rains, it Rains*, and supported by the Humanities division. The piece is an 8-channel work representing rainfall data from 8 weather stations in the area surrounding Sarasota, Florida in musical form. In addition to the electronic performance, an element of improvisation is present from live instruments, either synthetic or acoustic; the musicians are both onstage in front of the audence and scattered among and behind them to match the speaker positions. Since the piece is about rainfall surrounding the Sarasota area, the audience is essentially on a map centered on the city, and each speaker is positioned such that the sound originates from the direction of its source data on the map (for example, the Bradenton weather station is in front of the audience as it is north of Sarasota; the Venice station to the South belongs behind). Each data source, then, would output to a different channel (and, in the performance, a different speaker).

The initial prototype was presented to me as a SuperCollider patch consisting of a simple SynthDef defining the sound to be made, as well as some glue code. Notably, also present in the source code was the complete raw data of a single channel to be sonified. The data had been copied and pasted into the file and then formatted by hand to fit SuperCollider's array format. In order to perform the entire piece, every channel's sound needed to be generated manually offline and then combined before the performance. For each of the 8 channels, the source code must be modified: the old data taken out, the new data pasted in, and then reformatted from CSV to a SuperCollider array. Finally, the program is run and the output is recorded; this must be repeated for all 8 channels every time a change is made to any of the sound synthesis parameters. Thus, it is immediately obvious that the version of this program in SuperCollider is not without its drawbacks.

This sonification project was well-suited to an experiment in the usefulness of music programming languages and related toolkits for several reasons:

- First, the program needed to generate the sounds was simple enough to be represented in any of the languages examined in the previous chapter. It also interacted with some of the common strengths and the weaknesses of many of the languages.

- Second, the piece at large had already been composed (although much of it was auditioned during the composition process through the aforementioned SuperCollider prototype), and so it allowed the sonification programs to be written to a spec and compared with each other. Often, when composing for electronic music, decisions made during the composition process are focused on the capabilities (and lack thereof) of the toolset in use, much like when composing for any other instrument. As a result, finding an existing piece that challenges the toolsets or at least fairly compares them to each other is difficult. *When it Rains* is for the most part a simple rule set and is therefore hypothetically somewhat language-agnostic.

- Lastly, the improvisatory nature of the composition provides grounds for comparing both music programming languages and other music programming contexts, like creating a fresh VST plugin and synchronizing it with the transport in a DAW. Music programming languages in particular are frequently cited as being useful for live coding, itself a potential improvisation tool, while running a native plugin has its own opportunities for facilitating improv as it has the ability to synchronize with other electronic elements in the outside world through the DAW as a center of musical activity, both in the form of external inputs and through other tracks embedded in the workstation.

Besides the prototype in SuperCollider and the native VST in C++, I examined several other potential languages as candidates and experimented to determine the sort of effort necessary in creating a minimum viable product for the composition in each language. If a language immediately presented difficulties in creating such a product that were corroborated by other programmers experienced in the language, the difficulties were noted and the prototype for that language was discarded. In the end, only the SuperCollider and VST forms of the project were practical for the core components of the project, although in all cases Max was useful as a glue between the rendered output, both off- and on-line, and the interfaces used for the performance.

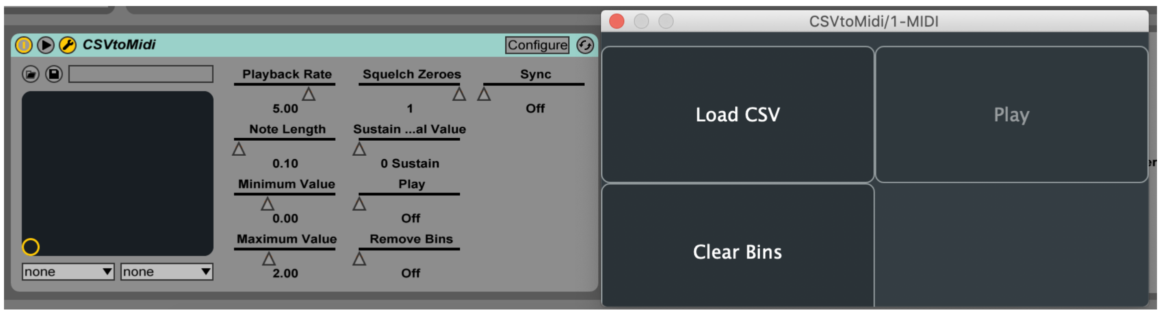
### 3.2.2 Choosing the VST

Reading the data was the primary concern when choosing a language to create the finished tool in. A CSV file is made up of lines of text. Most audio programming languages discussed in the previous chapter provide only extremely basic text file input/output capabilities; the difficulty of implementing operations as basic as iterating through a comma-separated file effectively ruled out using any music domain-specific language on its own. SuperCollider and Max are both difficult in this regard; ChucK on the other hand does provide input/output capabilities for text files. Similarly, the workflow used in music programming languages often results in programs with no user interfaces or (in the case of visual programming languages like Max) interfaces that are embedded into the code itself; in a performance context, the interface needed to be completely uncluttered and largely as foolproof as possible. Max in particular does provide options for building a user interface for patches out of the flowgraph itself in the form of a ”presentation mode” that can hide objects not important to performance and rearrange the remaining objects in a more useful or aesthetically pleasing layout.[4] However, the difficulties with simply parsing and using the CSV data in Max had already put it out of the running before performance was even taken into consideration. Finally, to use complicated synthesis options, presets, timelines, and other basic comforts, the tool would ideally be embedded in a DAW. Max again works well here, but only in Ableton Live; other DAWs are not supported. To meet this concern and to make use of basic programming concepts common in general-purpose languages (like file I/O), writing a plugin in such a general-purpose langauge with help from an audio library could easily be as useful or moreso than any of the music languages presented, and indeed writing the plugin ”from scratch” ended up being the most promising and adaptable of all the prototypes considered. I programmed the tool as a VST audio workstation plugin in C++, using the JUCE library, which provides tools for audio processing, simple user interface drawing, and can output the dynamic libraries needed for a VST.[5] The resulting VST operates inside of the audio workstation and so gives the composer full access to the other tools used in creating electronic music, like use of the composer's existing synths, audio effects and all the tools needed for mixing and mastering, and critical timeline-based functionality like automation of parameters and VST controls. Loading a text-based CSV file in C++ is absolutely trivial; the VST approach also allows the plugin to be used alongside every major DAW using the specification. Additionally, generating notes in response

4. Cycling 74, "Max 7 - Presentation Mode," accessed March 27, 2018, `https://docs.cycling74.com/max7/vignettes/presentation_mode`.

5. ROLI, "JUCE — JUCE," accessed October 21, 2018, `http://juce.com`.

to events is somewhat better-suited to an imperative language-based approach than the stateless and functional paradigm used by Max (see the language discussion in the previous chapter); the code to examine the items in the dataset, for example, is far simpler than the mess of resulting rectangles required in Max, and easier by far to read even for an amateur than the esoteric layout of SuperCollider's synthdefs.



Playback Rate 5.00
Squelch Zeroes 1
Sync Off
Note Length 0.10
Sustain ...al Value 0 Sustain
Minimum Value 0.00
Play Off
Maximum Value 2.00
Remove Bins Off

**Figure 3.1:** The User Interface for the completed *CSV-to-MIDI* VST. Left: parameters automated by the DAW timeline. Right: the CSV-loading dialog GUI.

### 3.2.3 The Completed Plugin: An Audio Histogram

**histograms** A histogram is a visualization: a bar graph that sorts the data set up into a number of bins based on value, with each bar representing the count of the items sorted into that bin. The sonification plugin acts as a sort of audio histogram, first sorting the data into a number of bins. But instead of outputting the count of each bin all at once, the plugin emits a MIDI note for each data point received in the same order over time. This allows the nature of sonification to shine. Using the rainfall data as an example: when playing back the MIDI data into a synth, the sounds provide a sense of the amounts of rainfall that occurred over time based on the pitches playing at that moment rather than the static slice given in a graph. This is effectively ordinary quantization, but additionally maps each bin to a MIDI note and provides additional tools for specifying where each bin begins and ends.

**the histogram plugin** The completed sonification plugin acts as a MIDI effect. The user adds the effect to the track and loads the CSV with the desired data inside. The effect accepts MIDI input through the track: received notes create bins for each data point to fit into and map that bin to the received pitch. The plugin has controls to assist mapping the notes to the data: the maximum and minimum values of bins are initially set to the highest and lowest values found in the CSV file, respectively, but can be tweaked or automated with the DAW tools through the corresponding VST parameters. Each note received maps to a progressively higher bin, which allows the mappings from data to pitch to be non-linear; the artist (for example) might input a scale or arpeggio, or map lower values to the middle of the spectrum and higher ones to pitches at the top or bottom of the keyboard so that they stand out more.

For event-based sonification where each note should represent something happening, zero values or values at or below the minimum value provided can be squelched to prevent continual note triggers that only represent absence. For instance, a day with zero rainfall might have its note squelched- musically, a rest results instead of a synth "ting" that might be confused for a small amount of rainfall on its own. Sonification that represents data that changes slowly over time can make use of the "Sustain on Equal Value" feature that sustains notes instead of retriggering them when the next note would produce the same pitch. Temperature highs and lows, for instance, are a continuous value that changes relatively slowly over time (at least when compared to sporadic and sometimes abrupt rainstorms): the "Sustain on Equal Value" feature can keep a single note held to provide a synth pad texture until the temperature difference is enough to halt and sound a different MIDI pitch. Playback speed is controlled by setting the amount of time each note will play in seconds, or alternatively by making use of the DAW's timeline synchronization and emitting notes at regular (quantized and synchronized) intervals, eg quarter notes. In this way, the plugin can emit the notes completely asynchronously on-demand, or synchronize its playback with the rest of the DAW. Besides the note triggers, the position in the data that's being notated is synchronized with the transport, allowing for multiple sonifications to be synchronized and played back, or for other non-sonification sound sources to be mixed in: either simply as an aesthetic addition to the mix, or for the sonification to play a background role in a larger musical production.

### 3.2.4 Performance, and a SuperCollision

Once finished, the plugin was used to continue drafting the piece and to audition potential modifications to the parameters generating the notes. The plugin's MIDI output needs an accompanying plugin (or plugins) to actually generate the sound; some time was spent devising a signal chain in the DAW to produce a similar sound to the one originally generated by the initial SuperCollider plugin. One of the greatest strengths of using a plugin to generate the notes in a workstation was the complete decoupling from other modules, but in this case, the same strength was to some degree its downfall. For the plugin-centric version of the piece to be useful for a live performance, then, it needed to be accompanied by a synth and effects chain for each climate station source or voice. Usually, this would be a natural part of the composition or production process in electronic music, but in this case the original sounds from the bare-bones SuperCollider prototype were preferred by the composer. The exact character of a reverb or even a simple synth instrument can be difficult to replicate on other platforms, as the exact implementation details of the algorithms used can be both difficult to match up together and to determine to begin with. At this point in development, the scheduled performance was between two and three weeks away; rather than taking the risk of not finding a satisfactory sound or exactly emulating the prototype's, the original prototype in SuperCollider itself was used for the performance. As the design of the prototype required each channel to be pre-rendered offline, this performance setup precluded any form of improvisation or realtime aspect to the notes generated from the data itself. The improvisatory element of the piece was therefore exclusively from musicians performing to the pre-recorded music.

# Chapter 4

# Even Further

## 4.1 Strengths and Weaknesses

Like any programming language, music programming languages have trade-offs associated with choosing any one over the other; a musician wishing to diversify their toolset will need some understanding of these trade-offs in order to choose a useful language. Coding style and preference for syntax can shape the decision-making process, but more often than not the presence or absence of some critical base feature set will necessitate the use of one tool over another. To some degree, there may not be room for perfection; programming is in many ways a proverbial art as much as a science. But the design processes of music programming languages, often built for immediate use as a personal tool and then later adapted for a larger audience (as is exactly the case for e.g. SuperCollider), may leave something to be desired for any end user programming with it. More planned languages, on the other hand, (like ChucK) show more thought into intended use cases but again have some common failings; both of these types of languages tend to exist as a black box, completely separate from the myriad of other tools at the electronic musician's disposal. In music, programming is not in any way the only allowable creation process, and so it should follow the same rules any other modern tool would.

The obvious candidate for improvement regarding most of these languages is to add an integration with a DAW. As helpful as programming can be, expecting a producer to dedicate their entire workflow to a language in order to use it is unhelpful and divisive. Besides simply acting as yet another module for synthesis, the ability to hook into the parameters of other plugins to control them is a well of potential. Max finds much of its purpose here; embedded within Ableton Live, it functions particularly well as glue between other modules, either for routing signals and MIDI directly or for more complicated automation of the parameters of other synths. But Max's functional style has its own trade-offs. While immediately useful for the aforementioned signal routing or for basic synth patching, signal flow through a graph is only one way to work with data. The flowgraph style of programming is, in a way, more declarative than anything: it prescribes a signal chain once and deals poorly with incremental changes. While some objects in Max do allow for an imperative programming style, it exists more as an awkward compromise in both language documentation and in practice. For use cases where the desired output is thought of more as a series of steps than a set of relationships, the programmer-musician may prefer an imperative language, although this again returns to options that cannot synchronize with a DAW. Even for describing signal processing, imperative languages can be a better choice, as many popular synthesis and effect techniques like pitch correction are better understood as an algorithm than as a function.[1]

1. Harold A. Hildebrand (Antares Audio Technologies LLC), Pitch detection and intonation correction apparatus and method (U.S. patent US5973252A, filed October 27, 1997).

## 4.2 On Future Work

What, then, would an ideal language for music look like? Assuming a professional context, integration into the workstation is now obvious; choosing the look and feel of the language itself less so. Visual programming has a friendly presentation and works well with the parts of digital music production that have an obvious visual metaphor, like signal patching, which is easily representable similar to its analog counterparts. In a way, the Max style of programming can be more useful than the traditional DAW signal flow representation, which instead is usually focused around "racks" of effects with fewer obvious representations of the signal's actual path throughout. But an imperative approach is also useful.

An ideal approach may separate the two: Max's visual style would be used for describing the signal paths globally in the DAW but also inside of custom modules for defining simple custom synths, while an imperative language would be used for the actual automation, e.g. generating notes and harmonies. The imperative language need not even be domain-specific: a common choice for implementing these features in workstations for other domains is to use an existing scripting language, like Python or Lua, with extensions as needed to interface with the workstation as a whole. For complicated synthesis algorithms (or even for complicated automation in general), plugins running compiled native code are still useful; interpreted languages are still relatively slow and are considered by professional audio programmers to be useful more for prototyping than for production use.[2]

2. Costas Calamvokis, "Mixing it up: Audio plugin development in C++ and Lua, Costas Calamvokis, JUCE Summit 2015," accessed April 4, 2019, `https://www.youtube.com/watch?v=kZe78gvvDVI`.

## 4.3 Final Words

Given the current availability of languages, a musician looking to add programming to their arsenal will likely need to compromise on their tool. To work alongside their existing workflow, Max can integrate into an audio workstation, but users are then restricted to a specific DAW. Alternatively, languages like SuperCollider can be made to render offline, and the resulting files can then be sampled in the workstation. For live collaborative performance on the other hand, ChucK's collaborative multitasking can be useful. But in many cases, a domain-specific language isn't at all necessary; several libraries exist for creating DAW plugins in general purpose programming languages (like JUCE). Music languages are perhaps less useful for creating finished works in production contexts as they are in experimentation. The greatest strengths of these environments are found less often in the languages themselves, which have considerable competition with other general purpose languages, and more in their interpreters. The rapid prototyping (and indeed, even live coding) they enable is perhaps reason enough to use them on its own, and remains a feature unique to their niche.

# Appendix A

# Selected Works

Attached are compositions I wrote in the course of learning music programming languages. They can be useful in observing notable patterns in the languages used, discussed in Chapter 2, or enjoyed alone as examples of outputs thereof. All of the examples include a render in the form of a wav or mp3 file; the ChucK example is particularly useful for demonstrating improvisatory or live coding, and therefore the source code for the music itself has also been included.

- **Electrical Safety Seminar** - Ableton Live - *ch18.wav*
- **ChucK Piece** - ChucK - *chuck.ck, chuckthesecond.ck, chuckrender.wav*
- **Safe Testing Environment** - Ableton Live and Max - *safetestingenvironment.mp3*
- **Eidolon** - Ableton Live and Max - *eidolon.wav*

# Appendix B

# Code Listing

Described here is the source code for the sonification VST discussed in the third chapter of this document. A complete copy of the most up-to-date code can be found online at

`https://github.com/dot-operator/CSVtoMidi`

The code is built on the JUCE toolkit, which automatically generates a set of project files that already compile to a working VST synth or plugin (albeit one with no input or output). The autogenerated files, besides the project files in the root folder, include `PluginEditor.cpp` and `PluginProcessor.cpp` as well as their associated header files. Also in the project is `CSV.cpp`, which contains the framework for parsing the input files and for mapping them to their respective bins.

## B.1 Building

To build with Visual Studio, the pre-generated `.sln` file can be opened; the build settings are already configured to build both a VST and a standalone version of the plugin (essentially useless as the plugin only outputs MIDI data). To build for another operating system, eg. macOS, the JUCE launcher ProJucer is required; it can be downloaded from

`https://shop.juce.com/get-juce/download`

Open `CSVtoMidi.jucer` with the Projucer application and set the "Selected Exporter" drop-down at the top of the window to the preferred IDE. Click the icon to the right of the drop-down to generate project files and to open them in the IDE.

## B.2 Class Walkthrough

**CSV** A CSV object acts as the collection that holds the CSV files, an iterator for grabbing values therefrom, and tracks the collection of bins the values from the file will map to. Its primary interactions with the outside world are once through `loadCSV` and `getNextValue`. `loadCSV` takes as input a string and opens the file with that name, recording the last value of each column. `getNextValue` is essentially the portion of the iterator that performs the iteration, but it returns a MIDI note instead of the raw value from the CSV file, so it also bins the value to the correct note. The set bins themselves can be modified with `clearNoteBins`, `addNoteBin`, and `setBinMinMax`.

**PluginProcessor** The bulk of the plugin-relevant processing takes place in the PluginProcessor class. Autogenerated by the JUCE project are a series of overriding functions that provide basic information about the content of the plugin to the host, for example what name should represent it in the DAW. Some of these functions represent setting and getting *presets*, which are groups of parameters that can be browsed in the DAW and selectively restored. Since the plugin doesn't have any musical features on its own, using presets doesn't make much immediate sense. The functions mostly remain stubs; however, some DAWs expect at least one preset to be available. The plugin therefore exposes one preset, but doesn't name it or provide any functionality.

PluginProcessor mostly wraps the CSV class and attaches it to relevant VST features, including changing the CSV object's functionality from external parameters and grabbing notes from the iterator to send to output. Most of this takes place in the `processBlock` function. In an audio effect plugin, this function would receive an audio input buffer and perform the desired operations on it. As this is a MIDI plugin, the function instead simply acts as the main update loop of the program. Instructions from the plugin GUI, like commands to reset the note mapping entirely, are processed first. If the plugin is set to sync with the transport, the current playback position is compared to the DAW's, and additional notes are queued to play if necessary. If a MIDI note is received as input, the CSV object adds a new bin at the end that maps to that note. Lastly, any queued notes are generated using the CSV object's mapping. A Note Off message is also queued, and may also be sent directly in certain circumstances (eg., if the DAW transport is stopped).

**PluginEditor** The PluginEditor class tells the library what and when to draw in the plugin's settings window and communicates with the PluginProcessor object, a reference to which it receives on construction. The object attaches a listener to the buttons and responds by activating the corresponding functions in the PluginProcessor (either to Play, Load a CSV file, or to Clear the bins).

# Bibliography

74, Cycling. “if Reference.” Accessed March 18, 2019. `https://docs.cycling74.com/max5/refpages/max-ref/if.html`.

———. “Max 7 - Presentation Mode.” Accessed March 27, 2018. `https://docs.cycling74.com/max7/vignettes/presentation_mode`.

Aaron, Sam. “Sonic Pi License.” 2016. `https://github.com/samaaron/sonic-pi/blob/master/LICENSE.md`.

al, Aho et. *Compilers: Principles, Techniques, and Tools.* 2nd ed. Addison Wesley, 2006. ISBN: 0321486811.

al, Kapur et. *Programming for Musicians and Digital Artists.*

al, Wilson et. *The SuperCollider Book.* MIT Press, 2011. ISBN: 9780262232692.

Autodesk. “Python in Maya.” 2018. Accessed April 1, 2019. `https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2018/ENU/Maya-Scripting/files/GUID-C0F27A50-3DD6-454C-A4D1-9E3C44B3C990-htm.html`.

Cabrera, Andrés. “An Overview of Csound Variable Types.” *CSOUND JOURNAL* 1, no. 10 (2009).

Calamvokis, Costas. “Mixing it up: Audio plugin development in C++ and Lua, Costas Calamvokis, JUCE Summit 2015.” Accessed April 4, 2019. `https://www.youtube.com/watch?v=kZe78gvvDVI`.

Hildebrand, Harold A. (Antares Audio Technologies LLC). Pitch detection and intonation correction apparatus and method. U.S. patent US5973252A, filed October 27, 1997.

Hloover Sigurosson. “Overtone User’s Guide.” 2018. `https://overtone.github.io/docs.html`.

“Introduction — Cabbage Audio.” 2018. Accessed April 8, 2019. `http://cabbageaudio.com/docs/introduction/`.

List, SuperCollider Users Mailing. “SC vs Pure Data for teaching.” Accessed July 8, 2018. `mailto://sc%C2%ADusers@lists.bham.ac.uk`.

McCartney, James. “SuperCollider.” 2019. `https://supercollider.github.io`.

Nisan and Schocken. *The Elements of Computer Systems: Building a Modern Computer from First Principles.* 1st ed. MIT Press, 2008. ISBN: 9780262640688.

“Opcodes Overview.” 2019. `https://csound.com/docs/manual/PartOpcodesOverview.html`.

Puckette, Miller. “Max at Seventeen.” *Computer Music Journal* 5, no. 26 (2002): 31–43.

———. “The Patcher.” *Proceedings, ICMC. San Francisco: International Computer Music Association*, 1988, 420–429.

ROLI. “JUCE — JUCE.” Accessed October 21, 2018. `http://juce.com`.

Saunders, James. “No Mapping”.” *MusikTexte*, no. 149 (2016).

Software, Valve. “Soundscripts - Sound Operator Stacks.” 2011. Accessed April 4, 2019. `https://developer.valvesoftware.com/wiki/Soundscripts#Operator_stacks`.

Sutter, Herb. “The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling.” *Java Report* 5, no. 7 (2000).

University of California, Berkeley. “Solar Wind/Education and Public Outreach for STEREO/IMPACT and Wind.” Accessed July 11, 2018. `http://cse.ssl.berkeley.edu/stereo_solarwind/sounds_programs.html`.

University, Florida State. “Downloadable Data - Florida Climate Center Office of the State Climatologist.” Accessed October 21, 2018. `http://climatecenter.fsu.edu/climate-data-access-tools/downloadable-data`.

Wang, Ge. “ChucK : Strongly-timed, Concurrent, and On-the-fly Music Programming Language.” 2015. Accessed July 8, 2018. `http://chuck.cs.princeton.edu`.

———. “The ChucK Audio Programming Language: An Strongly-timed and On-the-fly Environ/mentality.” PhD diss., Princeton University, 2008.