SHIFT:

A LIVE CODING LANGUAGE FOR CLASSROOMS

By

NICHOLAS BROWN-HERNANDEZ

Source Code: https://github.com/combjellly/every1

Demonstation:https://youtu.be/behIQ-HMRuY?t=817

Try It: https://combjellly.github.io/every1/

supervisor: Dr. David Ogborn

a Major Research/Paper

Submitted to the Department of Communication Studies and Media Arts

in Partial Fufilment of the Requirements

for the Degree

Master of Arts

in Communication and New Media

McMaster University

Acknowledgement/Gratitude

This project could not have been possible without the help and guidance from the faculty and staff at McMaster University. It is important to remember that this project was formed & completed on the traditional territory of the Haudenosaunee, Attiwonderonk and Anishinaabe people. Being an uninvited guest, I recognize that my privileged position is shaped from years of systematic oppression of indigenous lives. My creative and academic work is complicit with these very colonial structures, as much of the work didn't research or investigate indigenous perspectives while also relying on systems that have and continue to oppress these nations. I also wish to express gratitude to those who housed me through this. Those who had patience to explain, articulate and share ideas that shaped this project & those who listened.

Preface

Quebec has recently passed a bill (Bill 96) that further implicitly oppresses those who currently do not speak French. This is an attempt to "protect Quebec values", values shaped from colonial violence. Though the bill may not pass entirely, it is emblematic that further resistance must be enacted to truly protect and acknowledge who's land we are located on. The linguistic component of this project (the fact that it is a coding language) relates to this as it is emblematic of core values that I have grown to embody (simplicity, community focus and multi-sensory communication models, etc.). However, the language also exemplifies problematic core values that have been nurtured through privileged bias such as, being only in English, relying on relatively powerful computing and being solely sound based. As the language evolves, it will be important to embody and recognize how development may or may not be perpetuating colonial ideologies.

Shift

The goal of this research project was to develop an accessible live-coding language that could be used in classrooms to facilitate the participation of students in collaborative music-making. The core function of the language is to allow students to control and alter the playback of sounds

using a computer. Providing students with a simple tool to learn coding and glean instantaneous feedback via aural cues, this language offers educators an alternative method of concurrently teaching programming and art simultaneously. This language is tentatively called *Shift*.

Shift's incarnation came from an inquiry into how live coding could be brought into as many different classrooms as possible. My experience teaching with already established languages that aimed to achieve such a goal was encouraging, showcasing that collaborative computer-based coding languages do have a place in educational spheres. The need to make this language wasn't to replace these established languages, but instead to provide educators with an alternative that more closely mimicked coding languages that would normally be taught in high school computer science classes. This model (in theory) would help collaborative live-coding find it's place in already established curriculum (Menard 2015; McCurdy, Schmiege & Winter 2008). Educators wishing to implement the arts into STEM (Science, Technology, Engineering and Mathematics) programs need to provide convincing arguments as to how these two culturally divergent disciplines demonstrate points of convergence (Yakman Lee 2012). The result of this would be that many different types of classes (computer sciences, music, art, math etc) would have an entry point to engage with media literacy in their curriculum. Currently, as outlined in UNESCO's 2013 document on pedagogical media literacy strategies (and later in their 2021 document) — "Teachers without formal training in media and information literacy (MIL) will be required to implement MIL programmes [found in the provided guidelines]" (Unesco 2013, p.112). These versions of the UNESCO publications on media literacy contain no mention of art making as a tool to develop media literacy nor do they mention how these guidelines can be adopted for non-media based classes around the world. Thus, much of the integration of media

literacy in classrooms remains static — often times prioritizing the analysis of media messages and less-so about the creation of such messages.

This paper will examine the pedagogical and philosophical elements that informed the design behind this language, as well as an investigation into literature that examines the value of collaborative live coding in classroom learning environments. Much of the decisions that percolated towards the top of this preliminary "alpha" build of the language have been informed from auto-ethnographic research during my 4 consecutive years teaching programming arts to high school students ranging from ages 12 to 16 at Heritage Regional High School. Investigations into whether or not these observations were unique and isolated to my own experience helped cement certain decisions, and will continuously shape this project over time. Thus, it is important to state that this project is ongoing and will be worked on even after submission of this MRP.

Design for Classroom

Guided by principles of social-constructionist theory, this project was designed from the ground up to be used by students collaboratively. Social-constructionist paradigms aim to provide a learning space for individuals to take part, and build knowledge as a community. Abandoning individual assertions of truth, social-constructionism seeks to build knowledge via social interactions - allowing for communities to construct their truth together (Andrews, 2012). In ensuring that this language promotes social-constructionist methods, it was vital that it was designed to be collaborative and easy to "pick up" for many.

A key design element is that it is geared towards people who may have never coded before. Educators will be encouraged to use this in classrooms, thus many choices in design stemmed from an importance on making it easy to learn (by virtue, also easy to teach). Bulger and Davidson note that a challenge facing the integration of media literacy into high school programs is insufficient resources allocated towards training teachers specific technological skill sets. Current efforts in training teachers (in the US) are primarily done via grassroots led initiatives, led by "impassioned educators" (2018, p.5). Thus, the language should be designed in a way that makes it easy for teachers to pick up and integrate into their own curriculum as they please. Expected challenges will be further discussed in this paper.

Student-Teacher relationship

Shift's simplicity gives opportunity to position the educator into a role of collaborator, removing the extreme need for them to lecture or walk around the room resolving/troubleshooting errors that students may run into. Instead, they can lead, conduct and prompt students accordingly. This shift in engagement with the class would have positive outcomes on student learning experiences. Skinner and Pitzer investigate the effect of strong student-teacher relations. From their research they found that direct teacher involvement with students work may strengthen the autonomy of students, reconstructing their class room involvement to be more self-motivated and less reliant on extrinsic motivation factors like grades — this is supported by self-determination theory (2012). Other theorists and researchers (Berndt, Hawkins, Jiao, 1999 ; Berndt & Keefe, 1995 ; Berndt, Laychak & Park, 1990) support this theory while also noting that bonds amongst peers may strengthen student motivations in classrooms. Freire and McCarthy also note that

these bonds may also encourage students to “feed off each others knowledge in order to expand their proficiency with different tools” when collaborating (2014 p.30).

Collaboration / Community / Listening

In order to encourage collaboration, the language was designed to give students enough tools to perform simple rhythmic structures. This requires that they collaborate with others to create larger, more involved compositions. The overarching simplicity that *Shift* has adopted positions each performer into a unique role that necessitates others to play with them. Individually, they can’t do too much. However as a group, they can create large, complex and evolving compositions. The language was designed in a way to acknowledge this trade off, and has been shaped to help make collaboration inviting and fun — even to those with no prior music or coding experience. Everyone who participates in an ensemble using this language is given the same set of limitations, promoting equal access. A democratization of sounds. In theory, this language will encourage classrooms to collaborate as a unit, instead of relying on students with high engagement to lead the class. This collaborative participation demands active listening amongst all participants, as well as forms of reciprocity in conjunction with collective collaboration.

Active listening, as discussed by Hildegarde Westerkamp (2019) and R. Murray Shafer (2009), provides an opportunity to stimulate, encourage and shape new ways of paying attention. In her book “Living as a Bird” Despret states that giving your attention to others, and understanding that others are themselves giving their attention to you and others is a means of acknowledging

importance. Acknowledging that others have a voice to be listened to and interacted with. Robin Wall Kimmerer has written a lot about this. Succinctly, she has written that “Paying attention is a form of reciprocity with the living world, receiving the gifts with open eyes and open heart.” (2013, p.222). This reciprocity through intentional collaboration echoes shifting trends in art education. This is no surprise when reviewing Dewey’s conception of progressive education. He claims that there is an intrinsic link between art, society and communication. He theorizes that incivility comes from dividing humans into non-communicating sects of assumed identities (Dewey, 1934). The issue of non-communication can be remedied through art making as a tool to understand others. As Dewey explains: “[art is expressive, it] speaks an idiom that conveys what cannot be said in another language” p,211. This conjures the notion that art can be a socially constructive epistemological tool, and therefore can be used to bridge connections and understandings of communities that are socially and/or geographically separated from one another. As David Ogborn puts it, collaborative live coding provides an opportunity to share a space of “fluid co-presence” (Ogborn 2016, p. 27). Participants are offered with an invitation to listen and tune their awareness towards their involvement in constructing a “public entity”.

Other academics have also noted that live coding helps foster communities and builds connections both in and out of school. Roberts, Allison, Holmes and Taylor explain that Gibber (a web-based live coding language) has been used to foster a variety of group-centred spaces (2016). As an efficient tool to form clubs, guide classrooms and create ensembles, Gibber has been used to provide youth with an opportunity to engage with peers and involve themselves in creative extra curricular activities. Involvement with group activities inside and outside school

has shown to positively affect student peer relationships, a factor that is directly related with dropout rates in secondary school (Dimitras-Zorbaz, 2017).

In prioritizing collaboration, it was important to build the language from the ground up with the expectation that it would be networked and available online for people to use. The language has been constructed in such a way that it should be easy to make it available within the Estuary platform for collaborative live coding (Ogborn et al 2022).

Web Focus

In order to facilitate later availability into Estuary, the language was coded in Purescript - a purely functional programming language that can natively compile to Javascript. Working within a Javascript context allows for the simple integration of code into web browsers, as web browsers have the ability to execute Javascript code on the user's device without any need to install external software. This is because web browsers are a "safe" environment that are designed to act as a sandbox, where code can be run without interfering with "root level" processes outside the browser. As a by product of this, *Shift* requires no installation to run making it readily available to any one who has access to the Net. This framework lends itself to easily integrate into classrooms for two reasons: (1) Teachers aren't required to have proper permission to install new software on school computers. Often times any software installed on school computers must be authorized by technicians or IT specialists, this could take very long depending on the school's protocol. Having everything accessible via a web browser ensures that classrooms may be more predictable for the teacher, resulting in less prep time and in-class troubleshooting; (2) Because of its integration into the web browser, it is easier to implement

networking capabilities. These capabilities simplify collaborative coding both in and outside of the classroom. It also enables remote collaboration, so that ensembles may perform/jam together anywhere. COVID-19 exacerbated the necessity for remote learning options. *Shift* and other live coding languages compatible with Estuary provide a fun and engaging way for classrooms to network over distances.

Though there are plenty of upsides to this, there are a few drawbacks. One, noted by Ogborn in his 2022 presentation at the Web Audio Conference, is that the web audio API "bottlenecks" computation potential of computers (Ogborn et al 2022). Being restricted to the browser "sandbox", results in audio becoming distorted when low powered computers are under a lot of "computational stress". This is further exemplified when there are many people performing at once, as they are all feeding the computer simultaneous requests to playback audio. This will prove to be a complicated limitation to deal with as many classrooms may have ten or more students performing at once. Relying on powerful computers may be a difficult requirement for this language to find it's ways into a variety of different schools, especially if the school only has access to low powered devices. Once the language is operational and deployed, it will be important to research how well it performs on a variety of computers in different classroom settings. I theorize however that due to the limited scope of performance (how simple the rhythms can be are per individual performer), this problem may not come to light unless there are large classes performing at once.

Process

The development of this language was divided up into 4 sections. (1) Designing how the language will look and how it will sound (2) coding a parser that is able to read user input, then produce an abstract syntax tree. (3) create a rendering engine that is able to time events from parsed input (4) integration of webDirt, a sampling engine developed and maintained by David Ogborn.

Design

Prior to coding a parsing algorithm, design concepts were drafted and worked through via conversations with Dr.David Ogborn. These concepts were guided by the design philosophies discussed earlier, ensuring that the language was simple to understand without prior music or coding experience was a necessity. Following in the footsteps of other languages such as P5JS and SonicPi, *Shift* was crafted in such a way that it closely imitates the way that current popular coding languages are structured. As exhibited in his Dagsthul seminar on Live Coding, Mark Guzdial notes that students appreciate exposure to syntactically "relevant" coding languages that extend beyond educational tools (Blackwell, McLean, Noble, Rohrhuber 2014). Exposure to these "relevant" patterns provide students with an understanding of how industry standard coding languages operate. The reasoning for basing one language off of other more prominent languages is exacerbated by Margolis' observations that "disadvantaged youth may be keenly aware of educational experiences that deny them technical mastery" p.80 (Aaron, Blackwell, Pamela 2016). This may affect participation and engagement with the language. Thus, this language will be designed to allow users to build programs using vertical looping structures found in languages such as Python and Javascript. It is important to acknowledge that these languages are both

imperative in it that they require explicit instructions to operate. *Shift* is designed to be more declarative, as it it defines what should be occurring in each loop. These differences are trivial for the moment as *Shift* is still in its infancy and exhibiting qualities that are commonly found in both these two paradigms of language design.

Another underlining design structure that this language exhibits is that it's not frustrating to use. Due to the tactility of typing, live coding places an emphasis on kinetic motor learning (Chwirka 2002). When designing a language that primarily uses a keyboard as its input, it is important to recognize that not all keyboards are the same, thus the demand of motor learning will vary from class to class & student to student. Geographical variations of language force keyboards to represent the regional language. Due to this occurrence, some keys (tokens[1]) require many more keystrokes to access (often times using two hands at once to). Common examples of this are " **: { } ~ $ #** , which all require holding down the shift key. This language based its design, syntax and form around this — focusing its parsing separators on line breaks and tabbing to reduce student frustration when searching for a key on a keyboard that they may not have quick access to. As a side effect, this language can be used by inputing one character at a time.

Finally a design element I've integrated into this project is that the code should be self explanatory. The language should be designed in a way that embodies a combination of cybernetic epistemologies and portions of Thomas Green's "Cognitive Dimensions of Notation" — imploring that there should be as little guesswork as possible to reach a desired outcome. The code should thus simply represent what effects are output from its rendering system. "Cognitive

[1] A token is a symbol that the coding language uses to represent an element of a coding language.

Dimensions of Notations: Design Tools for Cognitive Technology" (2001) clearly summarizes and discusses the framework that Green designed in the 90s. This framework describes this key design element as "Closeness of Mapping" — where the notation should describe the result it produces. To put it simply, it should be clear what affect the code has on the sounds that are heard, by virtue — it should be easy to predict what needs to be changed in the code to have a specific desired output (role-expressiveness). Interaction with the code should be seamless and easy to engage with, essentially, accessible to many. This accessibility may be facilitated via the implementation of a closed loop learning schema that, provided they are given enough information to experiment on their own, students may construct their own knowledge via causal feedback loops between them and the output experience. As discussed in the 2001 paper mentioned above, each one of these design choices has a dimensional quality to them, implying that they are designed via a series of decisions that push the language into specific applicative directions. In doing so, the language commits to having specific characteristics that will have their trade-offs.

Design Trade Offs

Described as having "Orthogonal" features, Green and associates acknowledge that the 14 dimensions of cognitive notation each point towards distinct directions of design priorities. Trade offs in this context thus refer to "situations in which one source of difficulty is fixed at the expense of creating another type of difficulty" p.6 (Blackwell1 et. co 2001). Where in prioritizing one dimension, another loses focus. In the framework of this language, these trade offs can be grouped into two categories (1) those that relate to the syntax, structure and

understanding of the language and (2) those that relate to the possible results that one may achieve from the language.

(1) Syntactically, the language is designed in a way that prioritizes readability. This means that the language should be easy to "understand" without much previous programming experience. In doing so, the language does take up a lot of space to describe very simple rhythmic elements. The trade off here is that the program that the user will code will require a lot of input to achieve something relatively simple. This can give rise to simple syntax errors caused by accidental keypresses/token usage. Also, this is the reason that the language relies on multiple people to use it at once. This may prove to be difficult for a student to use on their own, or a teacher to incorporate into small classrooms if they wish to achieve complex results.

(2) This language operates within a declarative paradigm, being constructed in a style that expects the user declare/define what the sound is and when it occurs in time. To increase understanding of rhythmic metrics, the language requires that the user assigns a time of sound occurrence before they declare what sounds will play during that time. This is done using an "Every" token, where the user declares that "every 3(beats). play sound". Compared to other music oriented coding languages like Super Collider or Tidal, this over simplification of rhythmic structuring actually makes it harder to create complex sequences of sounds without doubling the length of the program. Green would refer to this as having "high viscosity" as the language has inherent boundaries to how adaptable the code is and

how resistant it is to change (Green 1990). Large portions of the code will need to change (or double in size) in order to have significant changes on the output experience. Languages like Tidal have low viscosity. Instead of having to create an entire new loop to complicate rhythmic structures, they enable the user to simply position sounds within a list (sequence). The language then interpolates the difference in time between each element contained in the list and produces a series of events that then playback sounds accordingly. High viscosity in a live coding environment enables for faster changes to the code, and by virtue, the resulting sounds. *Shift*, having high viscosity, implicitly slows down the process of changing sounds. This will have affects on both the listening and creation of sounds as it is informally known to be a "slow" language.

Syntax

Pictured below (figure a) is an illustration that depicts two "programs" coded in *Shift*. Each one has a ruler underneath it representing the timing mechanisms that are expected from the code. In metered musical expression, events are timed to an underlining tempo. This tempo (usually) is a marker for equally spaced time events. The ruler represented below indicates a span of time, divided into equal divisions (marked by the number of beats above each line). These divisions could be thought of as seconds (or any other arbitrary amount of time). The program encapsulated in the black box dictates when the sounds "bd" & "sn" should occur on this timeline.

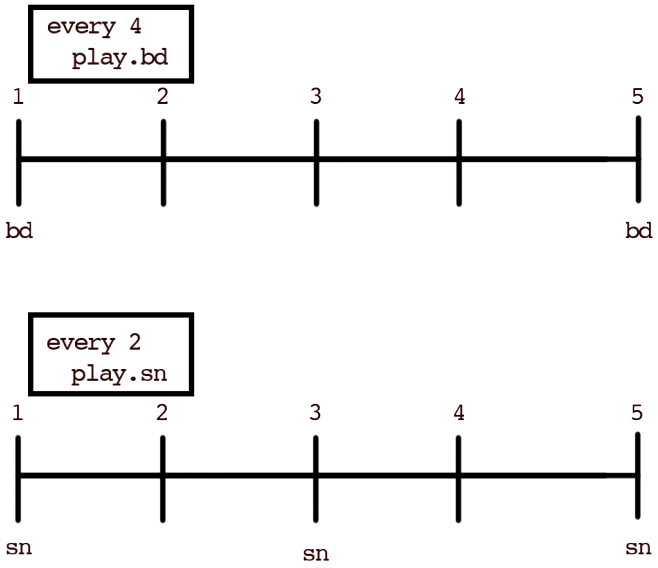


figure a

The first example shows that the sound “bd” is played every 4 beats, while the second example shows that the sound “sn” should occur every 2 beats. It is important to note that these two programs synchronize on the fifth beat. And so, if each student was making a program in this fashion - all their programs may synchronize and desynchronize over time giving way to musical syncopation (or, that thing that gives beat-based music it’s rhythmic complexity)!

Variables

Variables (in both mathematics and computing) are items that can be named and defined to contain a a series of numbers and/or expressions that may be used by different components of the program. These variables, sometimes looking like **“x=20”, “y=90 + x” , “dog = x * y”** in *Shift*, are programmed in a way so that they are evaluated every time the program runs through the Every loop they are contained within. Evaluation in this scenario is the process of defining and calculating something. These operations resolve in the order they are sequenced, and can be used to apply effects onto the sounds that are being played back. In the 2002 study conducted by

Milne and Rowe, they gleaned from an online survey that function and variable declarations are the easiest thing to teach and learn in programming. There is no reason given, however this exhibits perhaps the ubiquity of these functionalities in most coding languages. Sonic Pi and P5JS also have variable declarations, acting as elegant means of animating/morphing sounds over time. The inclusion of variables also brings with it the ability to include arithmetic, bringing in deeper and more complex musical possibilities. This shift towards variable controlled sample playback mechanisms shifts the language towards something more dynamic, allowing for samples and sounds to change and distort over time.

Parsing

A parser is a piece of code that reads text (or any other data) and produces an interpretation that dictates how a program acts. Essentially, it converts user input (in this case “every 50 play.sound”) and converts it to an Abstract Syntax Tree that feeds the program with rules dictating how it should output sound.

The previous examples demonstrate how the code may be parsed in its simplest form. It was important to ensure that any number that is parsed, may also be parsed as a variable. This would ensure that any number can change over time, allowing the user to implement dynamic controls over sound, rather than leaving them static.

To reduce complexity, each Loop and Action is separated by whitespace. White space is the empty space between characters resulting from indentation, line breaks or “spaces”. Certain conditions must be met however for these white spaces to properly separate sections. Majority of

this language was coded using PureScript, therefore, the implementation of its Parsing library[2] helped facilitate the creation of a parser that easily constructed error messages in case someone input code that didn't meet certain conditions. For one, "Every" must be followed by a number or variable or else the user is presented with an error that looks like " **(ParseError "Expected identifier" (Position { column: 7, index: 6, line: 1 }))**". This error message is constructed in a way to indicate to the user that the program expected something more to properly function. In another scenario, "play" requires that the user inputs a "." followed by the name of the sample they want to play. An error resembling "**(ParseError "Expected \".\"" (Position { column: 15, index: 14, line: 1 }))"** is presented to the user if they do not follow these structural requirements. If the program that the user inputs is syntactically correct, the parsing engine returns its results to the Rendering Engine.

Rendering Engine

The Rendering Engine is tasked with using the parsed content to create a list of events. Each event contains it's expected time of occurrence and a name of a sample. The time of occurrence is measured in Posix Time (also known as Epoch 1970 or Unix Time). The rendering engine is called every 1/4 of a second to verify whether or not the user has changed the program. Regardless of whether they have changed anything or not, the rendering engine takes the number assigned to the periodicity of the event (the number that follows "Every") and it provides a list of

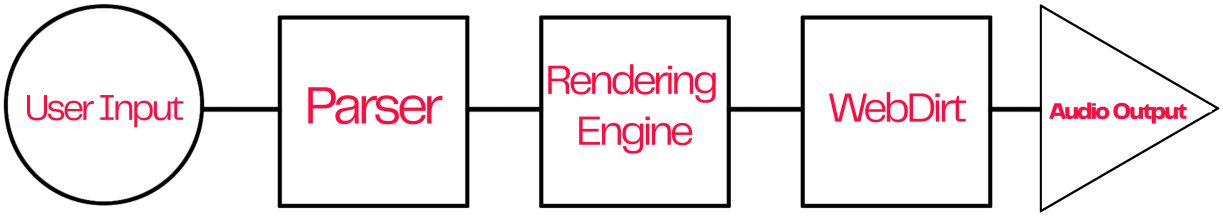
---

https://pursuit.purescript.org/packages/purescript-parsing/10.0.0

moments in time that that event should occur. This new list is then fed into WebDirt[3], a framework for outputting and playing audio events.

WebDirt

WebDirt is a library developed and maintained by David Ogborn to simplify the playback of pre-recorded samples. Recently it has been translated into Purescript, facilitating the integration of it into this project. WebDirt receives the audio events referred to above (constructed as {sample_name, Posix_time_of_occurrence}) and sends a request to the WebAudio API to play the audio file being referred to in the event. WebDirt contains a solid toolset to construct and integrate custom sample libraries, allowing for sounds to be customized and referenced without having to use any external libraries. WebDirt also contains a series of tools and functions that allow for sounds to be processed and distorted. Including control over these effects and processes will be one of many additional features integrated into *Shift* in later builds once variables are fully operational.



This figure represents the flow of data between User Input and Audio Output

---

https://github.com/dktr0/WebDirt

Future Revisions

As noted earlier, there are additional features and implementations that will be worked on after the submission of this project.

Conditional Operation

With the inclusion of variables, conditional operations would provide teachers with enough of a solid foundation to dedicate various lessons to this language alone. Conditional operations are common amongst programming languages. They are statements that compare 2 variables (or a variable and a number) and allow the user to program a set of parameters that change depending on the relationship between the 2 numbers/variables. Python and other object oriented programming languages use “if” and “else” statements as shown below. The program shown compares x to 4, if it’s less than 4 than it increments its value by 1, if it is more than 4 then it resets to 0. The idea would be to implement this into an “Every” loop, so that every time the Every loop signals an event, it would also evaluate conditional operations having some effect on the running program.

```
if x < 4:
        then x = x+1
else:
        x = 0
```

Future Research

All this being said, I think that this tool simply provides another entry point to research and experiment with live coding in classrooms. Affects on media literacy, classroom dynamics and community building would all be valuable topics to look into further using this language as a method to research and construct curriculum. Research questions that would be worth investigating would be the following,

(1) How can collaborative music making with code help students socially construct knowledge?
(2) How can skills developed in a live coding environment be reapplied in other media spheres?
(3) What effects may collaborative live coding have on student peer relationships
(4) What effects may collaborative live coding have on teacher-student relationships
(5) What effects may a "slow" live coding language have on active listening and compositional structures?

Furthermore, as stated at the very beginning of this - it is important that this language is accessible to as many as possible, regardless of language familiarity and exposure. As this project progresses forward, it is imperative that translation functions are included to ensure that the language can be used without relying on english as the sole entry point. This would take time and perhaps a deeper connection to community to get translation requests. Bringing me to my final discussion point, this language will always be designed with educators and classrooms in mind. Therefore, it is essential that feedback from teachers is discussed and implemented moving forward. I understand that my experiences do not line up with other educators, and so, moving

forward I wish to investigate how this language may be tuned to be more efficient, clear and fun for use in classrooms.

Bibliography

Aaron, Samuel, Alan F. Blackwell, and Pamela Burnard (2016). "The development of Sonic Pi and its use in educational partnerships: Co-creating pedagogies for learning computer programming." *Journal of Music, Technology & Education* 9.1: 75-94.

Andrews, Tom(2012). "What is social constructionism?." *Grounded theory review* 11.1.

Berndt, T. J., Laychak, A. E., & Park, K. (1990). Friends' influence on adolescents' academic achievement motivation: An experimental study. *Journal of Educational Psychology,* 82 , 664–670.

Berndt, T. J., Hawkins, J. A., & Jiao, Z. (1999). Infl uences of friends and friendships on adjustment to junior high school. *Merrill-Palmer Quarterly*, 45 , 13–41.

Berndt, T. J., & Keefe, K. (1995). Friends' influences on adolescents' adjustment to school. *Child Development*, 66 , 1312–1329

Blackwell, Alan F., et al. "Cognitive dimensions of notations: Design tools for cognitive technology." *International conference on cognitive technology*. Springer, Berlin, Heidelberg, 2001.

Blackwell, Alan, Alex McLean, James Noble, and Julian Rohrhuber (2014). Collaboration and Learning through Live Coding: Dagstuhl Seminar 13382. *Dagstuhl Reports 3:9*, pp. 130–168

Bulger, Monica, and Patrick Davison (2018). "The promises, challenges, and futures of media literacy." *Journal of Media Literacy Education* 10.1: 1-21.

Chwirka, Becky & Gurney, Burke & Burtner, Patricia. (2002). Keyboarding and Visual-Motor Skills in Elementary Students: A Pilot Study. *Occupational therapy in health care*. 16. 39-51. 10.1080/J003v16n02_03.

Dewey, J. (1934). Art as experience. New York, NY*: Berkley Publishing Group.*

Dimitras-Zorbaz, S. (2017). School Engagement of High School Students , 42(189), 107-119.

Freire, Manuelle, and Erin McCarthy (2014). "Four Approaches to New Media Art Education." *Art Education*, vol. 67, no. 2, pp. 28–31., https://doi.org/10.1080/00043125.2014.11519262.

Green, T. R. G. (1990) The cognitive dimension of viscosity: a sticky problem for HCI. In D. Diaper, D. Gilmore, G.

Cockton and B. Shackel (Eds.) Human-Computer Interaction — INTERACT '90. Elsevier.

Kimmerer, Robin Wall (2015). Braiding Sweetgrass. *Milkweed Editions.*

McCurdy, Sandra M., Cindy Schmiege, and Carl K. Winter (2008). "Incorporation of music in a food service food safety curriculum for high school students." *Food protection trends* 28.2: 107-114.

Menard, Elizabeth A (2015). "Music composition in the high school curriculum: A multiple case study." *Journal of Research in Music Education* 63.1: 114-136.

Milne, Iain, and Glenn Rowe (2002). "Difficulties in learning and teaching programming—views of students and tutors." *Education and Information technologies* 7.1: 55-66.

Ogborn, David (2022). "Estuary 0.3: Collaborative audio-visual live coding with a multilingual browser-based platform". 2022 *Web Audio Conference*

Ogborn, David (2016). "Live coding together: Three potentials of collective live coding." *Journal of Music, Technology & Education* 9.1: 17-31.

Roberts, Charlie, et al (2016). "Educational design of live coding environments for the browser." *Journal of Music, Technology & Education* 9.1: 95-116.

Schafer,Murray. R (2019) ". Murray Schafer: Listen" (Video). Dir. David New, 6min. www.youtube.com/watch?v=rOlxuXHWfHw

Skinner, Ellen A., and Jennifer R. Pitzer (2012). "Developmental dynamics of student engagement, coping, and everyday resilience." *Handbook of research on student engagement*. Springer, Boston, MA. 21-44.

UNESCO (2013). "Media and information literacy: policy and strategy guidelines" 196p. https://unesdoc.unesco.org/ark:/48223/pf0000225606

UNESCO (2021). "Media and information literate citizens: think critically, click wisely!" 407p. https://unesdoc.unesco.org/ark:/48223/pf0000377068

Westerkamp, Hildegarde (2019) "The Disruptive Nature of Listening: Today Yesterday Tomorrow" in: *Sound, Media, Ecology. Eds* Milena Droumeva and Randolph Jordan, Palgrave MacMillan, pp 45-63.

Yakman, Georgette, and Hyonyong Lee (2012). "Exploring the exemplary STEAM education in the US as a practical educational framework for Korea." J*ournal of the korean Association for Science Education* 32.6: 1072-1086.