Omni and Alga: new DSLs for audio programming and live coding

Francesco Cameli

U1563275

University of Huddersfield

MA by Research (Music)

Date of submission: 22nd April 2021

Abstract

This research presents two new *DSLs* (Domain Specific Languages): one to develop audio algorithms and a second to order the structure of their connections over time.

The first one, *Omni*, allows users to define the behaviour of *audio objects* at the lowest level. In *DSP* (Digital Signal Processing) terms, *Omni* is a language that describes the *sample-by-sample* behaviour of an algorithm. It compiles *Omni* code to native binaries for all the major operating systems (*macOS*, *Windows* and *Linux*) that can then be imported and used in creative coding environments such as *Max* or *SuperCollider*.

Alga, on the other hand, is a framework for *live coding* developed as an extension to the *SuperCollider* environment. Its main feature, in contrast with similar projects like *TidalCycles* or *FoxDot*, is not to define *musical patterns* whose changes happen *statically* only on code execution (however complex these changes are), but to use the act of manipulating code to describe the start of *interpolation processes* from *current states* to *future ones*, over specified windows of time.

This research is situated from the perspective of an electronic music improviser whose main motivation is to build these tools to enhance his creative practice. Therefore, development choices are both driven by personal aesthetic needs and the desire to have an integrated and fluid workflow in live performance scenarios (*Alga*) and low-level algorithm development (*Omni*).

The submission includes the source code for both projects, together with installation notes and usage examples. Furthermore, *x86-64* binaries for various *Max objects* and *SuperCollider UGens* are provided for all operating systems.

Acknowledgments

I would firstly want to thank my two supervisors, Alex Harker and Frédéric Dufeu. Their help has been invaluable not only during the course of this Master's degree, but also over my former studies as an undergraduate. Furthermore, I am deeply grateful for the open and diverse community of the *CeReNeM* at the University of Huddersfield, where I did grow not only as a student and researcher, but also as a person. Among my peers, I would like to mention the numerous insightful conversations I had the pleasure to share with James Bradbury, who I am thankful to.

I also want to thank my family and friends, who all pushed me to always give my best even the most difficult times. Lastly, I want to thank my girlfriend Giulia for her incredible kindness and support throughout a very tough year.

Table of Contents

Table of Figures

Chapter 1 - Introduction

1.1 - Background to the research

The initial idea for creating *Omni* and *Alga* was inspired by the need to develop an intimate and personal interaction with the machine from the perspective of an electronic music improviser. *Textual programming languages* and their interface, aimed both at algorithm development and *live coding* performances, proved to be, in my case, the most minimal and extensible form of interaction between musical intentions and their realization through computer generated sounds. *Alga* is a framework that leverages *SuperCollider*'s audio server to provide a flexible way of connecting modularized components together in real-time. *Omni*, on the other hand, supports *Alga*'s dynamism by allowing the development of static audio objects, compiled beforehand with a clean, minimal and extensible syntax. These objects represent the sonic building blocks that are structurally arranged with *Alga* in my work as composer and improviser. In fact, despite being two independent projects with their separate philosophy and goals, they both share a common reasoning that inspired their existence: my own musical practice.

Since the beginning of computer generated music and sound processing, developing audio algorithms has mostly been a technical task for programmers and engineers who know low-level programming languages such as *C* and *C++*. Audio programming also presents many demanding challenges that can at times diminish the accessibility and ease of understanding of the underlying code. However, with recent advancements in programming language development it is now possible to design tools that can satisfy both these requirements. Therefore, just as in other creative fields (Noble, 2009), *specific dialects*, or *DSLs*, have emerged to bridge between the complexity of machine code and its widespread use across the so-called *creative coders*; artists who use programming tools in order to shape their craft. *Omni* follows this trajectory, being it an *audio DSL* that can appeal to *creative coders*, more than software engineers and low-level programmers. In this regard, *Omni* is more akin to a project such as *Cycling '74's gen~ (Cycling' 74, 2011)* rather than other languages such as *FAUST* (Orlarey, Fober & Letz, 2009) or *SOUL* (Storer, 2016), which appeal to audiences of trained developers. Nonetheless, *Omni* can indeed be an effective alternative to such languages, as its minimal syntax does not preclude performance, being the code compiled down to highly optimized *C* code.

Alga operates at a higher level than *Omni*: the one of *audio environments*. These environments, that usually combine both visual and textual interfaces, do not often expose the *sample-level* operations directly but assemble and connect together pre-compiled *audio objects*. What this allows is an interface for rapidly prototyping audio algorithms and musical ideas without the technical overhead of designing specific low-level algorithms. Different environments for music making have been developed over the last sixty years, ranging from the early *Music* systems (Mathews, Moore & Risset, 1974), up until more recent software like *Max* (Puckette & Zicarelli, 1990), *SuperCollider* (McCartney, 1996) and *PureData* (Puckette, 1997). Within these environments, which provide the basic audio functionalities, a number of other smaller *sub-systems* have also been established, each one with its own unique set of features. For example, languages like *ixi-lang* (Magnusson, 2011) or *bacalao* (Fraser, 2020) are cases of embedded *dialects* in the *SuperCollider* environment. On the other hand, frameworks such as *ppooll* (Filip, 2005), *FrameLib* (Harker, 2017) or *MuBu* (Schnell, Röbel, Schwarz & Borghesi, 2009), represent various endeavours to create *audio contexts* with their own specific syntax and functionalities, while still being enclosed in the experience of environments such as *Max*.

Alga falls in between being a *dialect* and a framework: it currently is a *SuperCollider* extension library for *live coding*. Its main feature is to define the connection and structuring of *audio nodes* as a dynamic process. In *Alga*, *d*efining a new connection via code execution does not apply the changes instantaneously, but it instead triggers a process of interpolation between a current state of the *node* to the newly defined one, over specific windows of time. The future aim is to provide *Alga* as a custom language, or dialect, that interacts with the core *SuperCollider* implementation: the research here presented is only comprised of the *SuperCollider* interface, which is already complete with all the features in the forms of usable *Classes*. In short, while the *AlgaLib* framework is already usable within the *SuperCollider* environment and syntax, the actual custom *language* on top of it has not been developed yet.

1.2 - An improviser's perspective

Ideally, the tools for improvising music, regardless of their hardware or software nature, should provide the players/users with optimal ergonomics in order for the interface to be easily accessible both in rehearsal and live scenarios. Programming languages, considering their textual interface, stimulate the performer not on a visual level, as it would be in environments like *Max*, but on a *linguistic* one: expressing musical intentions becomes an act that is deeply related to how the language is designed for both the human and the machine. The ability to rapidly communicate with the computer through text is a fundamental tenet of *live coding* languages and was one of the driving ideas in developing *Omni*'s minimal syntax and *Alga*'s 'plug and play' experience: ergonomics of interaction, when coming down to *DSLs*, are key factors for their successful usage. The quicker and frictionless the iterative process of coding and experimenting, especially with sonic algorithms, the easier it is to engage with the tools in performative scenarios, while also improving code readability and compactness. For my practice as improviser, these are essential features to bridge the interface between the human intentions and the machine interpretation in real-time (Brown, 2016). This process of 'thinking-in-action' (Cocker, 2016) allows me to establish a dialogue at different levels of complexity and abstraction: from the lowest algorithmic one, via *Omni*, to the highest structural one, thanks to *Alga*.

1.3 - Aims and objectives

A common aim for both projects is the accessibility and ease of use of the software. In the case of *Alga*, this approach leads to quickness in its employment in performative and rehearsal scenarios. Regarding *Omni*, the result is a smoother development process of *DSP* code. Interacting with an agile interface to explore sound programming proved to be, in the case of my own practice, not only a welcomed addition to quickly prototype ideas, but a necessary requirement to enhance the feedback process between initial thoughts on specific sonic algorithms and their realization; from the sample-to-sample behaviour in *Omni*, up to the musical arrangement in *Alga*. Furthermore, the development of *Omni* and *Alga* is strictly open source: their conception does not come from trying to fit the projects into commercial aims of the current pro-audio market, but from the personal need of specific features that were not present or easily accessible from other software.

While common guidelines influenced both projects, they certainly present objectives and designs of their own. For *Alga*, the fundamental aim is to propose a novel way of structuring real-time algorithmic improvisation. In contrast with the widespread approach of pattern manipulating languages, *Alga* shifts the attention of the user to the act of *patching audio nodes*, similarly to hardware modular synthesizers. However, the *patching* is not instantaneous, but it activates a process of interpolation from the current state of the system to the newly set one. With this solution, *Alga* proposes a *fluid* style of composition and improvisation, where no abrupt changes transpire unless explicitly prompted by the coder.

Omni's main objective is to simplify low-level audio programming by providing a minimal and extensible syntax; easy to learn for beginners, yet deep enough for expert *DSP* coders. On a linguistic level, *Omni*, like *gen~*, aims to be a language for creative coders, where the syntax is designed not to get in the way of the developer, abstracting away many of the issues that one has to face when dealing with audio programming, while not renouncing on performance. Additionally, *Omni* also provides users with the depth and flexibility of languages that are aimed at engineers and expert audio programmers, as would arguably be the case for projects such as *FAUST* and *SOUL*. Among other features, one that stands out is the presence of a robust *object-based* system that allows coders to define custom and re-usable *structures* and *functions*, which is a fundamental trait to develop software in a modular way. Finally, *Omni* is developed to be a cross-platform and cross-environment language. In fact, it is available for all major operating systems (*Linux*, *macOS* and *Windows*) and, thanks to its *wrapper* approach that is later explained, can be used to create objects for common audio environments like *Max* and *SuperCollider*. Another aim for *Omni*, then, is to allow users to write their code once and have it working the exact way in multiple contexts.

Furthermore, an aim for both is to allow other creative coders to be stimulated and inspired to challenge long-standing practices with a new technological perspective. In my practice, several software projects have performed a similar role. In this regard, *FrameLib*'s novel approach to frame-based audio processing or *gen~*'s ease of use to design *DSP* algorithms stand out. It is my hope that *Omni* and *Alga* can have a similar impact on other people's methods of coding, composing and improvising.

Chapter 2 – Methodology

2.1 - Software development through practice and iteration

I have been using both software projects heavily since their conception. Therefore, during the development process, their features and goals have been reviewed several times, following what I felt it was needed for my musical practice. The process of iterating between the coding of the software and its actual real-world usage has been one of the core development motifs: iteration and practice, just as in any standard musical training, have proven to be incomparable tools of discovery. *Alga*, in particular, not only augmented a constant feedback exchange between the development and the practice, one influencing the other, but also provided a learning process resembling the one of an actual musical instrument. However, considering the digital nature of *Alga,* some distinctions ought to be made between the approach to playing physical instruments and the one towards *live coding* instruments. In the case of the latter ones, as the music theorist Andrew Goldman states, the absence of the *sensory feedback* of an acoustic instrument creates a disjunction between *human movement* and *improvisational decisions* (Goldman, 2019). This process, according to the researcher and developer Tim Sayer, generates a *difference* between the cognitive load that a musician undergoes when performing with traditional instruments and when performing with digital or *live coding* ones (Sayer, 2016). In such regards, practicing music with *Alga* provides the same *distance* between code execution and physicality as any other digital instrument and *live coding* language.

Different researchers, composers and performers have attempted to blur this discrepancy. A particular case is the one of the researcher and improviser Lauren Sarah Hayes, who proposes the usage of *haptic interfaces* for digital instruments in order to provide performers with physical feedback and resistance. This approach opens up many compositional and improvisational behaviours that would not be possible otherwise, allowing an improviser to develop a new physical relationship with the digital instruments (Hayes, 2014). Such an interface, for example, can be developed through *video game controllers*, as Hayes does for her piece *Figuring-Operated String*. In this composition, in fact, she uses a cheap *Mad Catz Gametrack* controller, which would usually be used for golfing videogames (Hayes, 2014, p. 147). Another example is the *Feedback Cello* by the University of Sussex's researchers Alice Eldridge and Chris Kiefer. They developed an electroacoustic instrument that leverages the acoustic body and strings of a cello to generate self-resonating feedback through transducers positioned in various places of the instrument. The transducers, then, are driven by a digital system developed in *SuperCollider* that *listens* and *responds* to the input, in a continuous feedback loop where the performer has an active role in interfacing with both the *acoustic* instrument and the *digital* counterpart (Eldridge & Kiefer, 2017). This sort of *human - instrument - computer* interface, then, essentially hybridizes the *nature* of the sound generation, while still providing the performer with a close *physical* response to interact with.

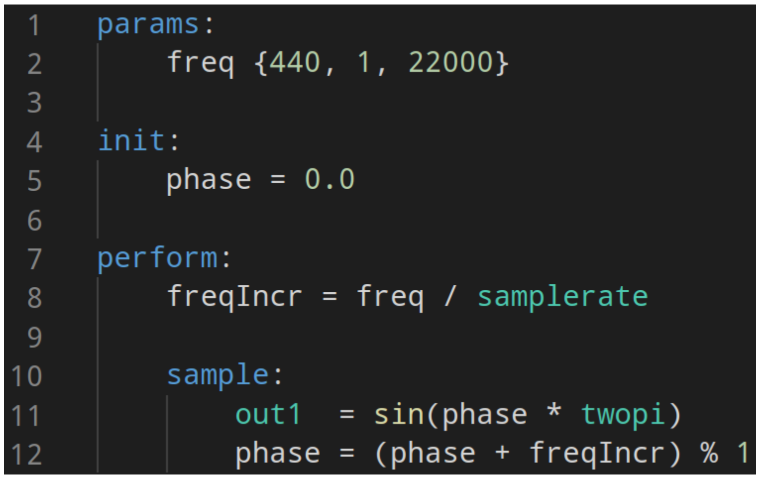
My approach to developing software is akin to the one of an instrument builder. Despite its physical disadvantages, I consider the computer as a *de-facto* musical instrument, as Max Mathews, the creator of the early *MUSIC* family of software, assessed in 1963 (Mathews, 1963). However, it is not just the *developed software* as a finished product that should be thought as such. In fact, as the creator of the *ChucK* language Ge Wang suggests, the act of programming itself - especially in the field of *live coding* - is to be regarded as musical instrument (Wang, 2008, p. 28). In my practice as coder and musician, the notion of *developing* the software and *playing* with it completely blur with one another. This mode of developing new tools differs from Miller Puckette's view of music software creation as a 'deadly embrace' between the developer and the user (Puckette, 2014, p. 8). Puckette addresses the two roles as being dependent on one another in order for each to be striving, technically and creatively. In my case, being in the peculiar position of both the *luthier* and the *player* allows me to merge the needs of either role in a constant evaluation of technical concerns and musical ones. The improvement of the software, then, does not only come from a pre-determined design scheme, but also, and most importantly, from the continuous creative usage of the tools themselves. Consequently, both the acts of *programming* and *using* the code influence one another in a continuous feedback process that is crucial in improving either aspect of my practice as programmer, composer and improviser. For example, using the software from the beginning of development permitted me to have the opportunity of experiencing *Omni*'s and *Alga*'s learning curves firsthand. During this process, I often found myself forgetting about the overall internal complexity of the system hidden behind the minimal syntaxes, inspired by the design of popular interpreted programming languages such as *Python* (Van Rossum, 2007), *Ruby* (Flanagan & Matsumoto, 2008) and *Lisp (*Steele, 1990)*.* In fact, I attempted to apply some of the features that make these tools appealing, even to newcomers: a clear and basic syntax that leads to prototyping quickness and rapid development of ideas, in contrast with the many pitfalls that *C-style* syntax presents to users, especially novices (Stefik & Siebert, 2013).

Opening up the development of *Omni* and *Alga* brought benefits that were only possible thanks to the open-source nature of the projects. This allowed both of them to 'stand on the shoulders of giants' by leveraging the open-source languages *Nim* and *SuperCollider*. *Nim* has been used to develop the entirety of *Omni*'s syntax via its powerful metaprogramming capabilities. By transpiling *Omni* code to *Nim* code, it is possible to build upon an already mature compiler infrastructure without the overhead and complexity that developing a custom one would take, especially for a single developer. Moreover, one more advantage is that *Omni* can be used for all the architectures that *Nim* supports, which, in turn, are the ones that *C* compiles to, as *Nim* compiles down to *C* code. On the other hand, *Alga*'s sound engine, *AlgaLib*, is coded in *sclang*, *SuperCollider*'s interpreted programming language. Using this language allowed my development to focus on the novelty of the implementation, instead of having to build a cross-platform sound engine from the ground up. By utilizing existing software, I was able to exclusively develop my ideas, instead of having to build the necessary scaffolding to tie all components together. Moreover, my hope is that integrating the project within a widespread environment such as *SuperCollider* will help with the adoption of *Alga*.

2.2 - Modular approach

In my work as a programmer I lean towards an abstraction based development process (Liskov & Guttag, 1986, pp. 9-17). This allows me to build programs by assembling together re-usable building blocks, promoting modularity. *Omni* presents the user with *code blocks* that represent the unique stages of an audio algorithm: definition of inputs and outputs, initialization of variables and processing of the audio loop. Coding with *Omni*, then, simply means to focus on *filling* these blocks with the necessary code to perform a specific algorithm, often without requiring the user of any general-purpose programming knowledge. Furthermore, *Omni* itself is built in such a way to promote re-usability of code across different projects: there is no need to duplicate the same *delay* implementation in two different codebases. Instead, it is possible to simply import and re-use the same *Omni* code in both. *Alga* exposes a different kind of modularity. Indeed, its method is more akin to the *modularity* that stems from the modular synthesizers' approach to sound generation. This approach arguably influenced how audio programming has been handled from its inception: defining individual modules that perform simple tasks, creating complexity not through the sum of the parts, but via the ways in which they interact with one other. In 1963, Max Mathews had already envisioned such an approach for computer music. In fact, his proposal was to allow composers to develop various instruments by interconnecting 'blocks of program that make up the instrument unit' (Mathews, 1963). *Alga*'s building blocks, *AlgaNodes*, draw inspiration from this modular technique: *musical complexity*, then, is not the result of *coding complexity*, but it comes from the intricate relationships established when *connecting* different algorithms together.

Chapter 3 - *Omni*: *DSL* for *low-level* audio programming



```
params:
    freq {440, 1, 22000}

init:
    phase = 0.0

perform:
    freqIncr = freq / samplerate

    sample:
        out1  = sin(phase * twopi)
        phase = (phase + freqIncr) % 1
```

Figure 1: A simple Sine oscillator in *Omni.*

3.1 - What is *Omni*?

Omni is a minimal and accessible programming language for audio programming, available for *Linux*, *macOS* and *Windows*. *Minimalism* and *accessibility* are generic terms to convey *Omni*'s goal of being easy to learn and to use, with few constructs to remember before writing productive code with it. The aim is for the *DSL* to have a rather flat learning curve: the advanced features of the language should not interfere with the creative coders' job, unless they have an explicit need to use them. *Omni* attempts to bridge the complexity of audio programming and its numerous moving parts with a clear interface that prompts the users on specific *blocks* they have to fill with their code.

The ease of use starts with the installation process. As long as *Nim* is installed, which is a very straightforward task on all operating systems, *Omni* can simply be set up via the *nimble* package manager with a one-line command: `nimble install omni`. Once this operation has been completed, the `omni` compiler will be available to the user. The `omni` compiler itself only builds a *shared* or *static* library with the optimized compiled code. This *OS-specific* library is then easily *embeddable* under any environment and API circumstances, even on-the-fly. The reasoning to compile code to *shared* or *static* libraries, then, is to enable any programmer to easily develop custom *wrappers* around the `omni` executable, which can act solely as the backend *DSP* compiler of a bigger project. Thanks to *Omni*'s simple and extensible language core and its *wrapper-based* approach, it is a very simple process for developers to expand *Omni* by writing new *wrappers* for specific projects. Personally, as a user of *Max* and *SuperCollider*, I have developed two wrappers for these environments, *OmniMax* and *OmniCollider*. Often, then, the creative coder would use the `omnimax` / `omnicollider` executables instead of calling into the `omni` compiler directly, allowing the same *Omni* code to be specifically *compiled* to either a *Max object* or a *SuperCollider UGen*. It is important to underline that the resulting binary is a *fully-fledged object* in the specific environment, completely integrated in the specific workflow.

3.2 - Contextualizing *Omni*

Omni is not the only available *audio DSL*. Similar projects have been developed over the last two decades: *gen~*, *FAUST*, and *SOUL* are all examples of languages specifically aimed at audio development and prototyping. Thanks to its ease of use, *gen~* can be regarded to be a tool aimed at *creative coders*, who would often use a low-level *audio DSL* to quickly prototype specific *DSP* ideas. On the other hand, *FAUST* and *SOUL*, due to their syntax and bigger scope of applications, are arguably more appealing to an audience of already trained *DSP* developers and engineers. Within this spectrum, *Omni* would be located in the middle, potentially being attractive for users on both ends.

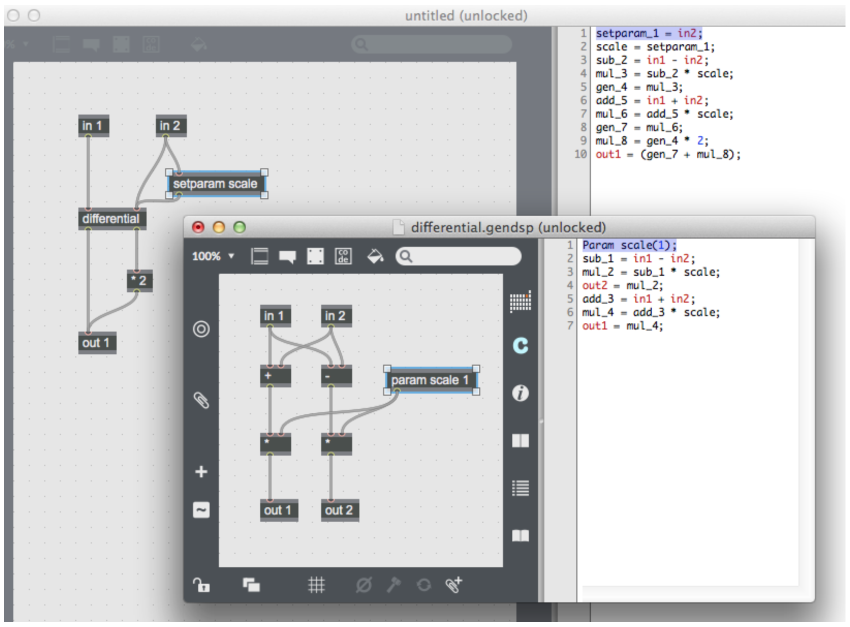
The appeal for beginner programmers and creative coders would be the minimal syntax that resembles *Python*'s, which is the gateway to computer programming for a high number of people, as demonstrated by being the fastest growing programming language in history

(Srinath, 2017). Regarding audio related tasks, *Omni* eases the way into complex topics such as *memory allocation / initialization* and *sample-by-sample* processing by defining those tasks in specific *blocks* of code that are *aware* of their context, and can therefore help the coders in their development journey. *Context-aware code* (Petricek, 2017) is here intended as a feature that prevents users from trying to force the system into doing actions that should not be done in audio programming, like allocating memory during the processing of real-time audio. For example, *Omni* allows coders to define instances of `structs`, which are its way of defining and allocating data structures, only in the `init` block, which serves the purpose of initializing specific variables to be used in the core of the algorithm. On the other hand, if the user attempts to create a `struct` in the `perform` block, which represents the *audio loop*, *Omni* will emit an error at compile time, hinting the coders that what they are trying to implement should not happen in the context of *DSP* development. As a result, newcomers would be intuitively informed of some common practices in audio programming while they are developing their own algorithms.

Some of *Omni*'s features might also appeal to expert *DSP* developers. First, *Omni* code is performant. In fact, despite its minimal syntax, which, for instance, does not require users to express variable types, *Omni* compiles down to highly optimized *C* code. As explained later, this is achieved by a type inference system that leverages *Nim*'s metaprogramming and code introspection to generate the fastest code possible. Furthermore, *Omni* provides a deep and extensible modular interface that promotes code re-use among different projects, allowing programmers to develop their own *libraries* to be imported and, perhaps, shared. *Omni*'s modularity, thanks to the `Data` construct that will be later described, also allows coders to have *dynamic allocation* of any type, including ones defined by the user. This feature is essential to quickly and reliably declare *arrays of* `structs` that are not bound by a set number, but can be dynamically defined in the `init` block via, for example, a `param`. This behaviour is described in section *3.5.3.1*.

Omni's syntax has been developed with the idea of being welcoming for any type of coder. Its goal is to provide a frictionless development process even for the most experienced *DSP* developers, helping them to focus more on the core of the algorithm rather than on specific syntactic complexities. Moreover, the multi-platform and multi-application compilation allows coders to develop their code once, being sure that it would work exactly in the same manner regarding of the operating system and environment in which it is being used.

Omni shares the same goal as other audio *DSLs*, which is to make audio programming more accessible and streamlined, simplifying the entry point to the core of the problem: specifying an algorithm that performs at the *sample level* of an audio buffer. Therefore, why did I choose to develop a new language instead of perfecting my understanding of existing ones? Simply put, no other tool provided me with a set of features best suited to my practice as developer and musician.

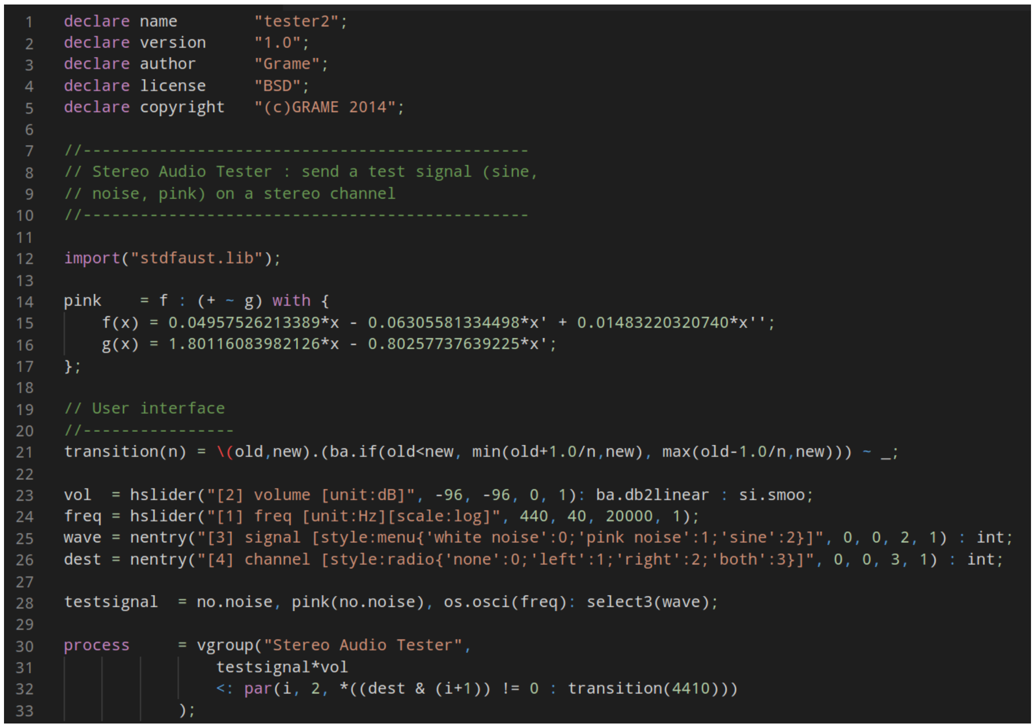


```
setparam_1 = in2;
scale = setparam_1;
sub_2 = in1 - in2;
mul_3 = sub_2 * scale;
gen_4 = mul_3;
add_5 = in1 + in2;
mul_6 = add_5 * scale;
gen_7 = mul_6;
mul_8 = gen_4 * 2;
out1 = (gen_7 + mul_8);
```

```
Param scale(1);
sub_1 = in1 - in2;
mul_2 = sub_1 * scale;
out2 = mul_2;
add_3 = in1 + in2;
mul_4 = add_3 * scale;
out1 = mul_4;
```

Figure 2: *gen~* code example.

gen~ is an innovative coding environment that proposes a single-sample patching system and language that only works within the boundaries of *Max*, which I am not a user of anymore. Besides, whilst it is exceptionally easy to quickly prototype ideas in *gen~*, it does not provide the programmers with the possibility of defining custom *data structures*, but only *functions* with many limitations to the scope they provide. In short, *gen~* only allows the creation of algorithms which are the combination of the *in-built objects*, without any support for expanding the palette of constructs to the developer's liking. The only workaround to achieve similar features is to wrap specific *gen~* code into re-usable functions, which might resemble some kind of object behaviour. However, this is limited by the fact that the only *in-built object* that is usable in functions is the `History` one, which simply performs a one-sample delay. All the other *in-built objects* must be declared and used solely in the global scope. This makes modularity very limited, as it mostly resolves around *copy / pasting* portions of code across different projects.

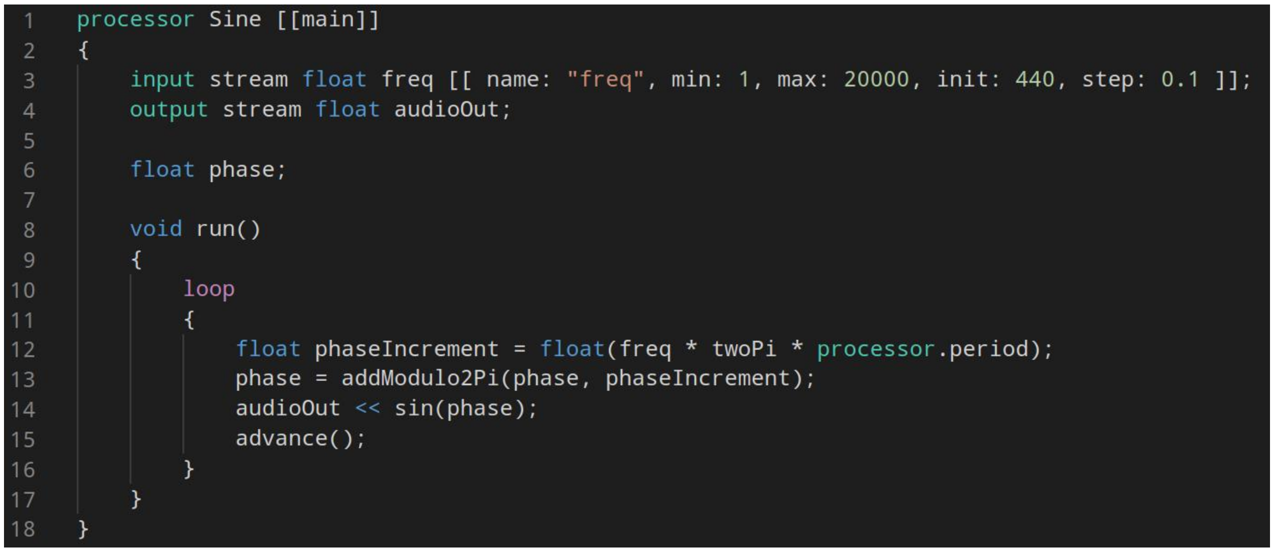


```
declare name        "tester2";
declare version     "1.0";
declare author      "Grame";
declare license     "BSD";
declare copyright   "(c)GRAME 2014";

//-----------------------------------------------
// Stereo Audio Tester : send a test signal (sine,
// noise, pink) on a stereo channel
//-----------------------------------------------

import("stdfaust.lib");

pink    = f : (+ ~ g) with {
    f(x) = 0.04957526213389*x - 0.06305581334498*x' + 0.01483220320740*x'';
    g(x) = 1.80116083982126*x - 0.80257737639225*x';
};

// User interface
//----------------
transition(n) = \(old,new).(ba.if(old<new, min(old+1.0/n,new), max(old-1.0/n,new))) ~ _;

vol  = hslider("[2] volume [unit:dB]", -96, -96, 0, 1): ba.db2linear : si.smoo;
freq = hslider("[1] freq [unit:Hz][scale:log]", 440, 40, 20000, 1);
wave = nentry("[3] signal [style:menu{'white noise':0;'pink noise':1;'sine':2}]", 0, 0, 2, 1) : int;
dest = nentry("[4] channel [style:radio{'none':0;'left':1;'right':2;'both':3}]", 0, 0, 3, 1) : int;

testsignal  = no.noise, pink(no.noise), os.osci(freq): select3(wave);

process     = vgroup("Stereo Audio Tester",
                testsignal*vol
                <: par(i, 2, *((dest & (i+1)) != 0 : transition(4410)))
            );
```

Figure 3: *FAUST* code example.

FAUST, developed at *Grame* in Lyon by Yann Orlarey, Stéphane Letz and Dominique Fober (Grame CNCM, 1982), is an open source project that, unlike *gen~*, provides users with a deeper development of custom *objects*, and it is distributed with a featureful standard library. However, I personally find its functional syntax hard to grasp quickly, as the grammar that it proposes is not easily relatable to any widespread programming paradigm. As a result, while *FAUST* code is extremely compact, the understanding of the algorithm on a *sample-by-sample* level is often rather difficult to comprehend. Finally, *FAUST* directly includes in its source code the support for a number of platforms and environments. In fact, when installing the `faust` compiler, a number of `faust2` additional commands are also bundled with it. The source code for all these wrappers, then, is embedded directly with the *Faust* one, which, in my view, leads the project to being rather *monolithic* for an *audio DSL*. For example, if one would only use the `faust` compiler to compile a *SuperCollider UGen* - thus calling the `faust2supercollider` command - all the other forms of export would be an overhead to that single use of the toolchain. Furthermore, this approach proposes a form of *centralization* of the project where the *FAUST way* of developing a compilation target with a custom `faust2` interface would be to submit the code changes to the main repository of *FAUST*. Then, once and if the changes are merged, these would now be *internal* to the project itself instead of being an individual *project within an ecosystem*.

Out of the three languages here proposed, *SOUL* is the more recent one, having reached its 1.0 release in January 2021. It is being developed at *ROLI* (ROLI, 2009) by Julian Storer and Cesare Ferrari. As a language, *SOUL* 1proposes itself to be the new standard in audio programming, especially in plugin and applications development. As a consequence, while its stated goal is to be easy to learn, a number of stylistic choices were dictated to appeal an audience of already trained developers, with a syntax that resembles *C*'s: the presence of brackets and semicolons, the need of declaring variable types, the use of the `void` concept, among other features. Let us consider the simple example of a sine wave generator, which is one of the simplest algorithms in *DSP* development:

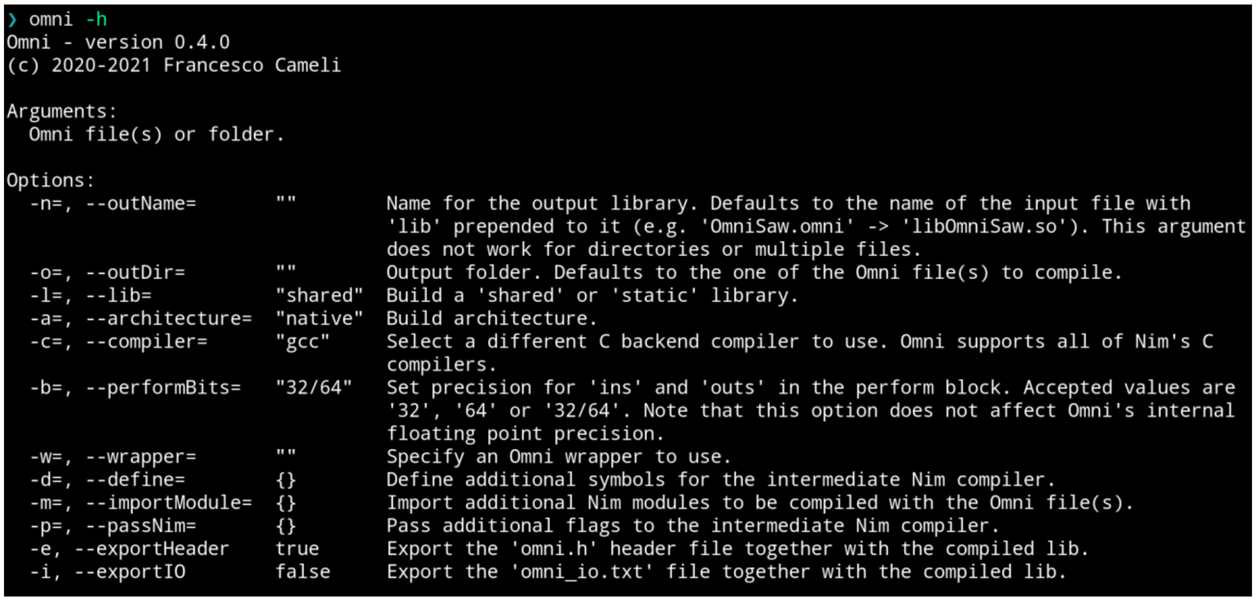


```
processor Sine [[main]]
{
    input stream float freq [[ name: "freq", min: 1, max: 20000, init: 440, step: 0.1 ]];
    output stream float audioOut;

    float phase;

    void run()
    {
        loop
        {
            float phaseIncrement = float(freq * twoPi * processor.period);
            phase = addModulo2Pi(phase, phaseIncrement);
            audioOut << sin(phase);
            advance();
        }
    }
}
```

Figure 4: Sine oscillator in *SOUL.*

As a newcomer with no programming experience, I would have to deal with some concepts that are adding complexity to such a simple example. Why would I need to express `void run()` considering that the `run` function must always be `void`? What is `void`? Why is `void` relevant to specify functions to execute an audio algorithm? What is the meaning of `void` for an audio programmer? Why is it required to manually advance the *sample-by-sample* loop by calling the `advance()` function? Where is this function declared? As an expert user I could argue that manually advancing the sample count would be practical to express different *rates* for specific components of an algorithm. For beginners, however, it can be perceived as a forced action that is required to be done, despite it not being directly related to the implementation being developed.

3.3 - The *omni* compiler CLI



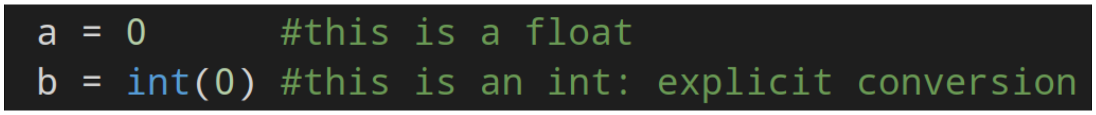
```
❯ omni -h
Omni - version 0.4.0
(c) 2020-2021 Francesco Cameli

Arguments:
  Omni file(s) or folder.

Options:
  -n=, --outName=       ""        Name for the output library. Defaults to the name of the input file with
                                  'lib' prepended to it (e.g. 'OmniSaw.omni' -> 'libOmniSaw.so'). This argument
                                  does not work for directories or multiple files.
  -o=, --outDir=        ""        Output folder. Defaults to the one of the Omni file(s) to compile.
  -l=, --lib=           "shared"  Build a 'shared' or 'static' library.
  -a=, --architecture=  "native"  Build architecture.
  -c=, --compiler=      "gcc"     Select a different C backend compiler to use. Omni supports all of Nim's C
                                  compilers.
  -b=, --performBits=   "32/64"   Set precision for 'ins' and 'outs' in the perform block. Accepted values are
                                  '32', '64' or '32/64'. Note that this option does not affect Omni's internal
                                  floating point precision.
  -w=, --wrapper=       ""        Specify an Omni wrapper to use.
  -d=, --define=        {}        Define additional symbols for the intermediate Nim compiler.
  -m=, --importModule=  {}        Import additional Nim modules to be compiled with the Omni file(s).
  -p=, --passNim=       {}        Pass additional flags to the intermediate Nim compiler.
  -e, --exportHeader    true      Export the 'omni.h' header file together with the compiled lib.
  -i, --exportIO        false     Export the 'omni_io.txt' file together with the compiled lib.
```

Figure 5: Help interface of the *Omni* compiler.

The `omni` compiler takes one file, a folder or multiple *Omni* files as positional arguments. Multiple flags are available to alter the result of the compilation. It is possible to change standard features like output name and directory (`--outName` and `--outDir`), together with compilation specific behaviours (`--lib`, `--architecture`, `--compiler`, `--performBits`). Moreover, flags for direct interoperability with *Omni* wrappers and the *Nim* intermediate compiler (`--wrapper`, `--define`, `--importModule`, `--passNim`) are provided. Finally, the `--exportHeader` and `--exportIO` flags can be used to export the '*omni.h'* and *'omni_io.txt'* files together with the compiled libraries. The former is used for *C* interoperability, while the latter contains metadata about inputs and ouputs of the compiled *Omni* code.

3.4 - Language design and implementation

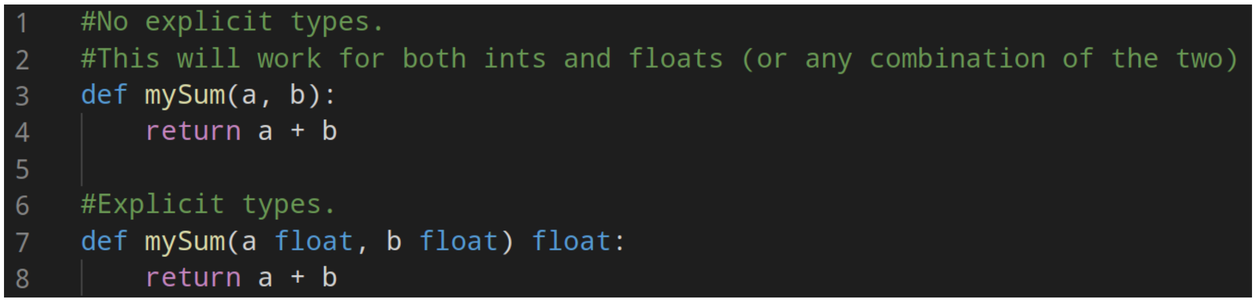
Omni draws inspiration from *Python*'s and *Nim*'s coding styles. From both of them, it borrows the indentation-based syntax, where specific scopes are determined by tabs and spaces, without curly brackets or semicolons. Regarding *Python*, I took inspiration from the quick approach to declaring variables, with no requirements of type specification or declarative syntax: it is only required to type a variable name and its value. The choice to go towards a *pythonic* syntax style was determined by the simplicity it allows the user to be interfaced with. *Omni*, however, is a fully statically typed language. In order to ease the work of the coder, *Omni* makes certain assumptions that are possible due to its context being limited to a *DSL* whose scope is solely audio programming. For instance, number types in *Omni* are defaulted to `float` (whose precision depends on the architecture of the machine being utilized), since this is the type that is most likely to be used in *DSP* to perform any calculation. It is certainly possible to declare *integers*, but this requires the user to be explicit about it:



```
a = 0      #this is a float
b = int(0) #this is an int: explicit conversion
```

Figure 6: Simple variable assignment.

Another example is the avoidance of explicitly setting argument and return types from a function, named `def` in *Omni*. Again, it is possible to state an explicit return type in case the coder is willing to enforce that specific behaviour:



```
#No explicit types.
#This will work for both ints and floats (or any combination of the two)
def mySum(a, b):
    return a + b

#Explicit types.
def mySum(a float, b float) float:
    return a + b
```

Figure 7: Two function definitions.

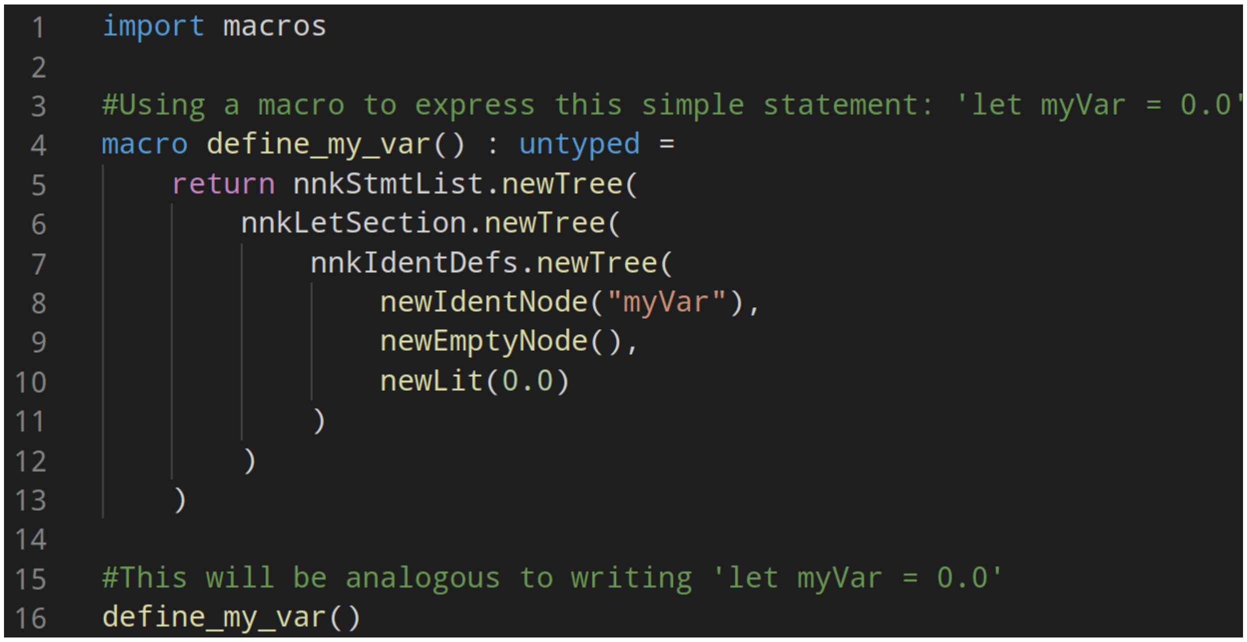
Once more, considering the limited scope of *audio programming*, a developer might tend to use the first case more, as it *includes* the second explicit one. This is not enforced by *Omni* itself, but it allows different types of programmer to feel comfortable with the system, allowing them to restrict their code to specific behaviours, if required. Other examples of *Omni*'s type system will follow in section *3.5*. The rest of *Omni*'s syntax is mostly inherited from *Nim,* yet with some exceptions. An example is the handling the constructor call for `structs`, which will also be analyzed later.

While *Omni* might feel like a dynamic language where no type information is required, its compilation results in strictly typed *Nim* code. This level of type inference has been made possible by the extensible *metaprogramming* capabilities that *Nim* offers. Finally, *Omni* is a *block-based* language. In order to compile a valid program, certain *blocks* of code must be filled with the specifics of the algorithm that is being implemented. Then, each *block* represents a specific function in the execution of the code. The analysis and usage of *blocks* will be explained in section *3.5*.

3.4.1 - Metaprogramming: choosing the Nim language

Creating a programming language requires the arrangement of several extremely complex components. From parsing and lexing the code, to representing its structure in the most optimal way with *ASTs* (Abstract Syntax Tree) (Thain, 2019, p. 83), down to generating native instructions for specific CPU architectures. While I regard myself as a fairly expert developer, undertaking the task of building an entire language infrastructure would take me years to realize. Yet, a number of modern programming languages provide options for *metaprogramming*. *Metaprogramming* allows the developer to alter the code generation of a program using specific constructs, often called *macros*. *Nim* (Rumpf, 2008), *Julia* (Bezanson, Karpinski, Shah & Edelman, 2012), *Rust* (Matsakis & Klock, 2009) and *Zig* (Kelley, 2016) are examples of languages that use *metaprogramming* as a powerful element to abstract many components of their implementation in *macros* executed at *compile time*. In short, 'metaprogramming is the process of writing computer programs that treat programs as data, enabling them to analyze or transform existing programs or generate new ones' (Lilis & Savidis, 2019). If the ergonomics of a language allow it, it is possible to develop entire *dialects* on top of a well-built *macro* system. This is what *Omni* does with *Nim*.

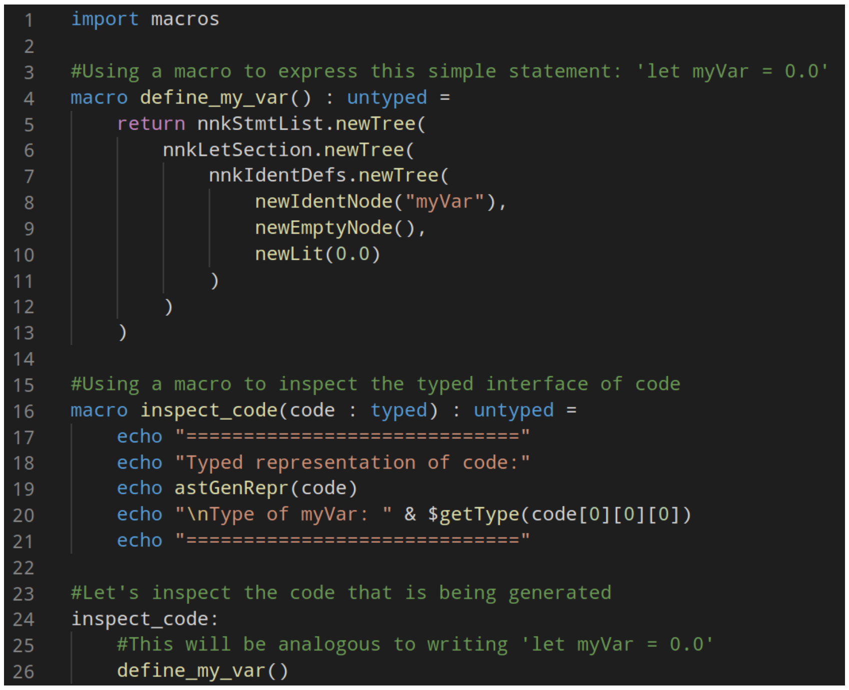
Nim provides an extremely powerful *macro* system that enables the programmer to directly manipulate the *AST* representing the code to produce a new valid *AST*. This allows the developer to implement custom parsers for the *AST* of any piece of code, as long that after their execution they would return a valid *AST* for *Nim*'s own parser. Furthermore, a feature which is extremely valuable to *Omni*'s implementation is *Nim*'s separation of the `untyped` code generation, which enables the *syntactic representation* of the code, from its `typed` counterpart, which includes the full type inference. Consequently, *Nim* provides different entry points in its parsing stages, which permits the programmers to inject their own logic within *Nim*'s code analysis. For clarity, let us consider the simple variable declaration `let myVar =  0.0`. The first stage, the `untyped` one in *Nim* terms, would be the simple representation of the code through an *AST*:



```nim
import macros

#Using a macro to express this simple statement: 'let myVar = 0.0'
macro define_my_var() : untyped =
    return nnkStmtList.newTree(
        nnkLetSection.newTree(
            nnkIdentDefs.newTree(
                newIdentNode("myVar"),
                newEmptyNode(),
                newLit(0.0)
            )
        )
    )

#This will be analogous to writing 'let myVar = 0.0'
define_my_var()
```

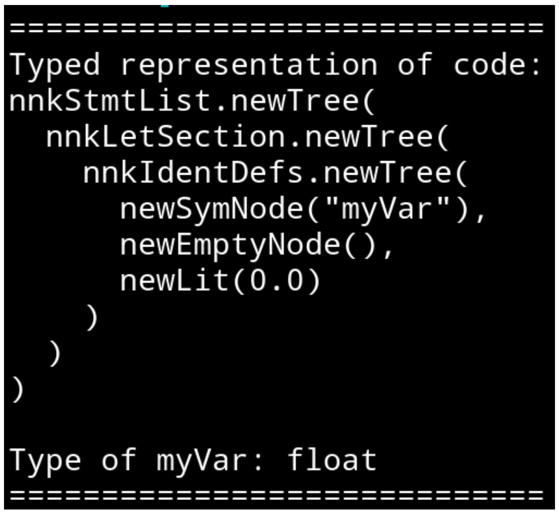
Figure 8: Basic usage of macros in *Nim.*

This *AST* is *untyped* because it does not contain any type information yet. The next stage, the `typed` one, will perform type inference on the *AST*, thus retrieving the type information for the variable `myVar`:



```
import macros

#Using a macro to express this simple statement: 'let myVar = 0.0'
macro define_my_var() : untyped =
    return nnkStmtList.newTree(
        nnkLetSection.newTree(
            nnkIdentDefs.newTree(
                newIdentNode("myVar"),
                newEmptyNode(),
                newLit(0.0)
            )
        )
    )

#Using a macro to inspect the typed interface of code
macro inspect_code(code : typed) : untyped =
    echo "============================="
    echo "Typed representation of code:"
    echo astGenRepr(code)
    echo "\nType of myVar: " & $getType(code[0][0][0])
    echo "============================="

#Let's inspect the code that is being generated
inspect_code:
    #This will be analogous to writing 'let myVar = 0.0'
    define_my_var()
```

Figure 9: Defining an extra macro to inspect the typed representation of the code.

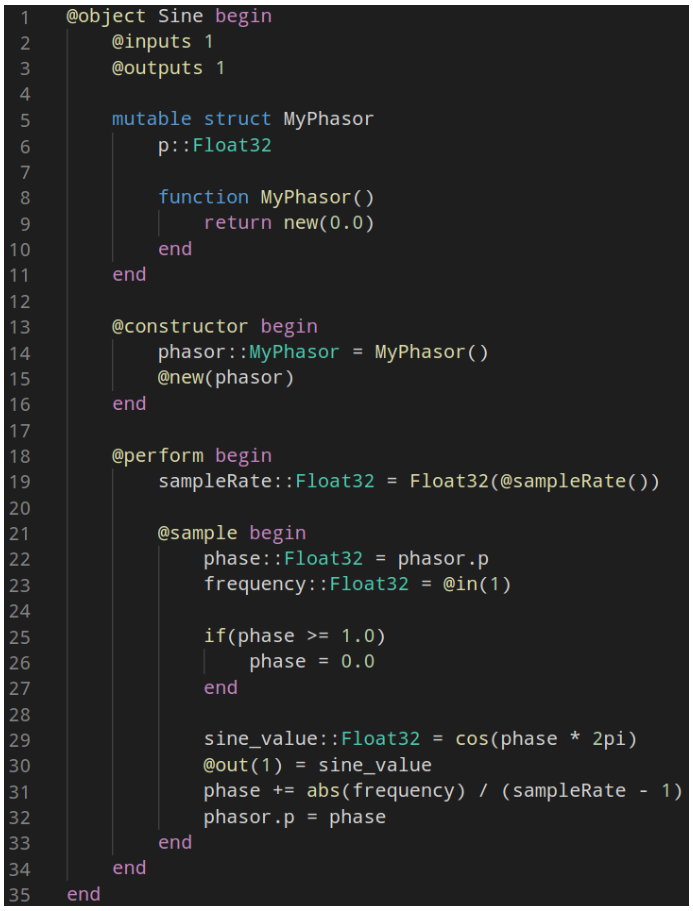


```
=============================
Typed representation of code:
nnkStmtList.newTree(
  nnkLetSection.newTree(
    nnkIdentDefs.newTree(
      newSymNode("myVar"),
      newEmptyNode(),
      newLit(0.0)
    )
  )
)

Type of myVar: float
=============================
```

Figure 10: Result of the previous code. All macros are performed at compilation time.

The difference between the two representations, in this simple case, only lie in the declaration of `myVar` as a `Sym` (`Symbol`) node, instead of an `Ident` (`Identifier`). If inspecting the `Symbol` node with the `getType` function, it would now be possible to see how *Nim* represents the `Symbol` belonging to that variable as a `float`. Through the usage of *macros*, then, *Nim* allows to modify both of the *AST* representations at compilation time. This powerful distinction allows programmers to inspect and build code that is *aware* of all the *type* context, enabling the injection of many optimizations directly in the code generation stages, without affecting the runtime performance of the compiled binary. *Omni* implements its own *parser* that acts on *Nim*'s *AST* on both of these levels, generating optimized *Nim* code which, in turn, produces *C* code, finally compiled to machine instructions.

Before using *Nim*, I have also experimented with other languages. My main project has been *JuliaCollider* (Cameli, 2019), which enables developers to use the *Julia* programming language inside *SuperCollider* in order to develop *algorithms* on the fly via *JIT* (Just-in-Time) compilation. *Julia*'s *JIT* engine, based on *LLVM* (Lattner & Adve, 2004), allows users to compile code very quickly and cache the results directly in memory, instead of writing them to a binary file. While I was successful in embedding *Julia* within *SuperCollider*'s real-time memory system, the project presented several other limitations. Stylistically, due to *Julia*'s requirement of prepending of the `@` symbol to *macros*, *JuliaCollider*'s syntax could not be addressed as a standalone *language*, which was my goal, but rather as a *macro sub-system* within a language. Moreover, *JuliaCollider* could not prevent the user from calling *Julia* functions that should have not been called in a real-time audio context, and the whole project relied on the assumption that the coder would not use the system the wrong way, with the possibility of crashes. Furthermore, the *JIT* compilation presented complications in terms of synchronization, and the overhead of *Julia*'s runtime and *garbage collector* was a huge burden with which I had to work around. Some of these problems were solved by directly forking the *Julia* language, and using this *patched* version of the system in *JuliaCollider.* Nevertheless, keeping up with the language development slowly put my forked version of *Julia* extremely behind the upstream one. While I would regard *JuliaCollider* as currently being a *deprecated* project, the development process has surely been an invaluable learning experience that allowed me to better design *Omni*.



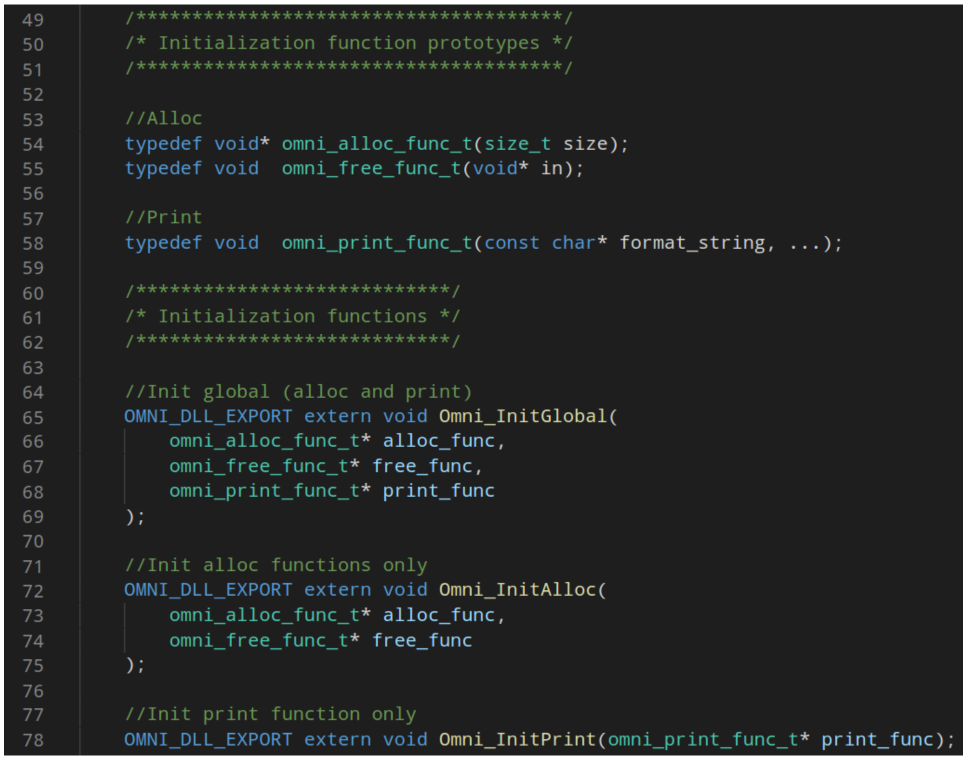
```
@object Sine begin
    @inputs 1
    @outputs 1

    mutable struct MyPhasor
        p::Float32

        function MyPhasor()
            return new(0.0)
        end
    end

    @constructor begin
        phasor::MyPhasor = MyPhasor()
        @new(phasor)
    end

    @perform begin
        sampleRate::Float32 = Float32(@sampleRate())

        @sample begin
            phase::Float32 = phasor.p
            frequency::Float32 = @in(1)

            if(phase >= 1.0)
                phase = 0.0
            end

            sine_value::Float32 = cos(phase * 2pi)
            @out(1) = sine_value
            phase += abs(frequency) / (sampleRate - 1)
            phasor.p = phase
        end
    end
end
```

Figure 11: A Sine oscillator in *JuliaCollider.*

3.4.2 - The exported interface

The `omni` compiler generates either a shared or static library containing all the compiled functions that are necessary to embed the specific *Omni object* into any project that provides a *C* compatible ABI or that can simply load shared libraries. This allows *Omni* to be easily wrappable by most, if not all, audio applications that already exist. The binary file is exported together with an '*omni.h'* header file that contains all the function definitions that compose an *Omni library*. There are two families of functions: `Omni_Init` and `Omni_UGen` ones.

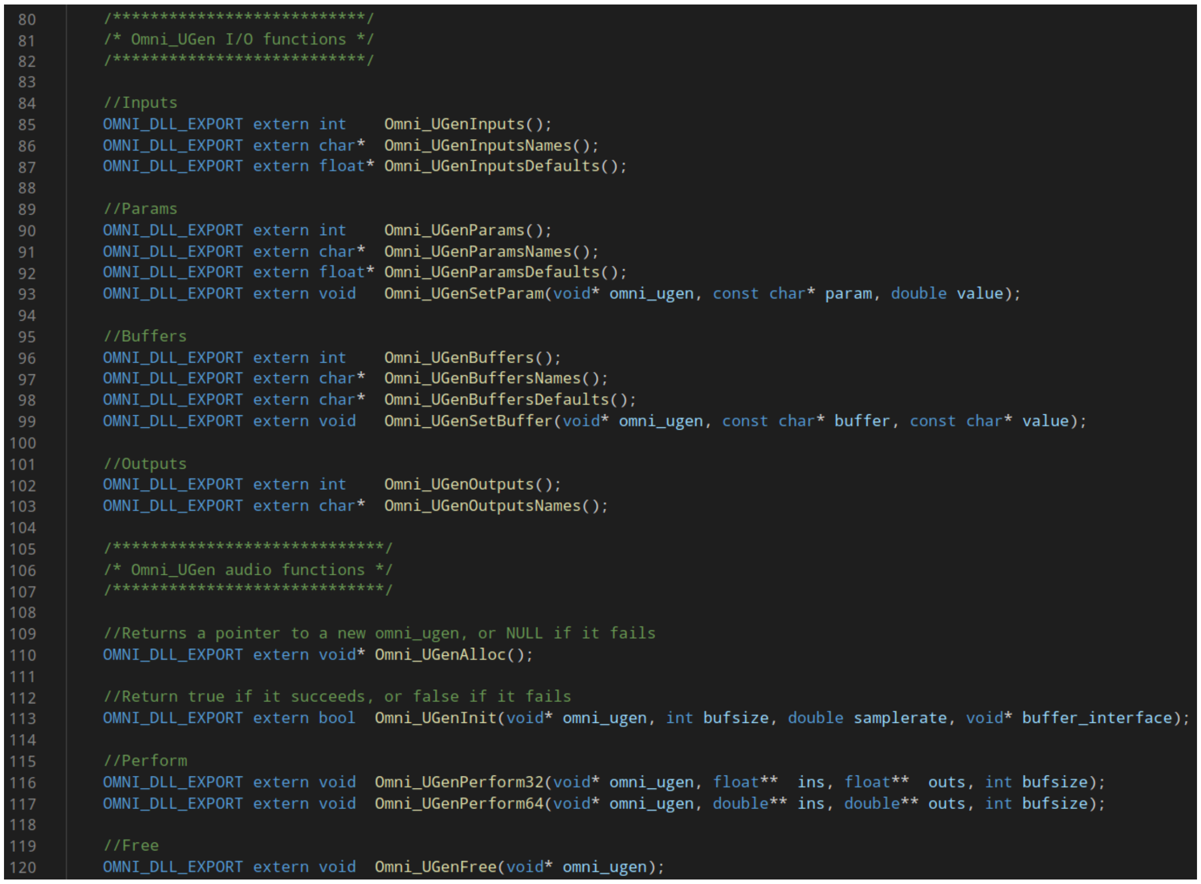
The first family allows the programmer to initialize *wrapper* specific features, like memory allocation and printing. *Omni*, in fact, enables the developer to specify custom `alloc` / `free` functions, permitting the library to be embeddable into any project, even ones that use a custom memory allocator. In the case of *OmniCollider*, for example, using *SuperCollider*'s real-time allocator for all of *Omni*'s allocations has been as easy as calling `Omni_InitAlloc` with the pointer to the specific `RTAlloc` / `RTFree` functions. The same philosophy applies to the `print` function, with the `Omni_InitPrint` call. It is important to specify that, if these two functions are never called, *Omni* will use the system's `malloc` / `free` and `printf` functions instead. Thanks to this interface, *Omni* provides the programmers that want to develop their own *Omni wrappers* with the possibility of specifying each aspect of the *Omni* execution, permitting them to embed the language in the most ideomatic way for their projects. I believe that, by giving access to such introspection, *Omni* should be flexible enough to work in any situation in the most frictionless manner, without the need for existing projects to adapt their interface in order to communicate with it.



```c
/****************************************/
/* Initialization function prototypes */
/****************************************/

//Alloc
typedef void* omni_alloc_func_t(size_t size);
typedef void  omni_free_func_t(void* in);

//Print
typedef void  omni_print_func_t(const char* format_string, ...);

/****************************/
/* Initialization functions */
/****************************/

//Init global (alloc and print)
OMNI_DLL_EXPORT extern void Omni_InitGlobal(
    omni_alloc_func_t* alloc_func,
    omni_free_func_t* free_func,
    omni_print_func_t* print_func
);

//Init alloc functions only
OMNI_DLL_EXPORT extern void Omni_InitAlloc(
    omni_alloc_func_t* alloc_func,
    omni_free_func_t* free_func
);

//Init print function only
OMNI_DLL_EXPORT extern void Omni_InitPrint(omni_print_func_t* print_func);
```

Figure 12: Initialization functions in ‘*omni.h’.*

The second family of functions deals directly with the audio algorithm. It includes global functions to retrieve information about the number of inputs, outputs, parameters and buffers. Furthermore, functions to allocate, initialize, execute and free an instance of an *Omni object* are provided. These are the functions that a *wrapper* would utilize to actually call into the core of the *Omni* developed algorithm.



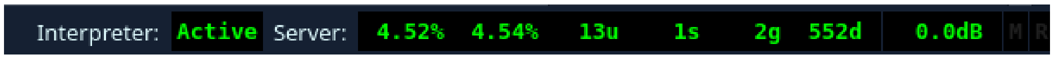
```c
/***************************/
/* Omni_UGen I/O functions */
/***************************/

//Inputs
OMNI_DLL_EXPORT extern int    Omni_UGenInputs();
OMNI_DLL_EXPORT extern char*  Omni_UGenInputsNames();
OMNI_DLL_EXPORT extern float* Omni_UGenInputsDefaults();

//Params
OMNI_DLL_EXPORT extern int    Omni_UGenParams();
OMNI_DLL_EXPORT extern char*  Omni_UGenParamsNames();
OMNI_DLL_EXPORT extern float* Omni_UGenParamsDefaults();
OMNI_DLL_EXPORT extern void   Omni_UGenSetParam(void* omni_ugen, const char* param, double value);

//Buffers
OMNI_DLL_EXPORT extern int    Omni_UGenBuffers();
OMNI_DLL_EXPORT extern char*  Omni_UGenBuffersNames();
OMNI_DLL_EXPORT extern char*  Omni_UGenBuffersDefaults();
OMNI_DLL_EXPORT extern void   Omni_UGenSetBuffer(void* omni_ugen, const char* buffer, const char* value);

//Outputs
OMNI_DLL_EXPORT extern int    Omni_UGenOutputs();
OMNI_DLL_EXPORT extern char*  Omni_UGenOutputsNames();

/*****************************/
/* Omni_UGen audio functions */
/*****************************/

//Returns a pointer to a new omni_ugen, or NULL if it fails
OMNI_DLL_EXPORT extern void* Omni_UGenAlloc();

//Return true if it succeeds, or false if it fails
OMNI_DLL_EXPORT extern bool  Omni_UGenInit(void* omni_ugen, int bufsize, double samplerate, void* buffer_interface);

//Perform
OMNI_DLL_EXPORT extern void  Omni_UGenPerform32(void* omni_ugen, float**  ins, float**  outs, int bufsize);
OMNI_DLL_EXPORT extern void  Omni_UGenPerform64(void* omni_ugen, double** ins, double** outs, int bufsize);

//Free
OMNI_DLL_EXPORT extern void  Omni_UGenFree(void* omni_ugen);
```

Figure 13: Functions regarding the state of an *Omni* object in ‘*omni.h’.*

3.4.3 - Technical analysis: *float*, function overloading and *Data*

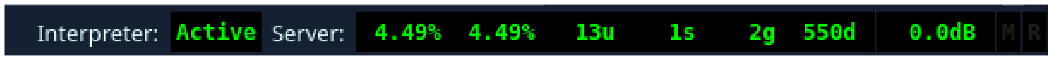
Before talking about syntax, it is relevant to highlight some of the issues that needed to be solved in designing the language.

On digital systems, audio is considered to be a *hard real-time scheduling* process, meaning that it has to be executed fast enough within specific deadlines in order not to produce *audio dropouts* (Boulanger & Lazzarini, 2010, chapter 3.4). In developing *Omni*, achieving a balance between performance and ergonomics has been one of the most important issues to address. Thanks to *Nim*’s introspection, for instance, it is not necessary to specify the type of variables while coding, as they will be inferred by the assignment operator. Types can either be numeric (`float` / `int`) or custom `structs`. Since *Omni* compiles to statically typed *Nim*, explicitly setting a variable type only helps with code clarity, but performance is as fast as when types are inferred by the compiler. Considering that most *DSP* algorithms operate with *floating point* precision, the `float` type is considered a first class citizen in *Omni*. This means that if a variable is inferred to be numeric, it will be defaulted to be a `float`, unless explicitly set to be an `int`. This assumption simplifies many operations, and it permits the code generator to optimize specific calculations and math functions by calling directly *C*'s *stdlib* counterparts under the hood. At the same time, if mixing `floats` with `ints`, *Omni*, thanks to *Nim*, has the notion of overloading the function calls depending on the type of the arguments. This feature produces another benefit. In *Omni*, it is possible to define a `def myFunc(a, b)` function that will work for any types of `a` and `b`, even if they are custom `structs`, granted that the operations in the body of the function are applicable to that specific type of the argument. I would like to mention that, due to timing, I have only scratched the surface in terms of optimizations: *SIMD*, stricter conversions, more inlined constructs are just few of the areas that could improve performance even more. Despite this, *Omni* still performs very quickly, as it is shown by this comparison between the same granular algorithm developed in *Omni* and in *C++*, both compiled to *SuperCollider UGens*:



```
Interpreter: Active Server: 4.52% 4.54% 13u 1s 2g 552d 0.0dB M R
```

Figure 15: CPU usage of granulator in *Omni.*



```
Interpreter: Active Server: 4.49% 4.49% 13u 1s 2g 550d 0.0dB M R
```

Figure 14: CPU usage of granulator in *C++.*

Another important behaviour that *Omni* takes care of is memory management. I wanted memory allocation to be transparent to the users, but at the same time I did not want them to have to take care of explicitly freeing it in a specific *block*. Memory allocation only happens when instantiating a `struct`, and it is only possible to do so in the `init` block.

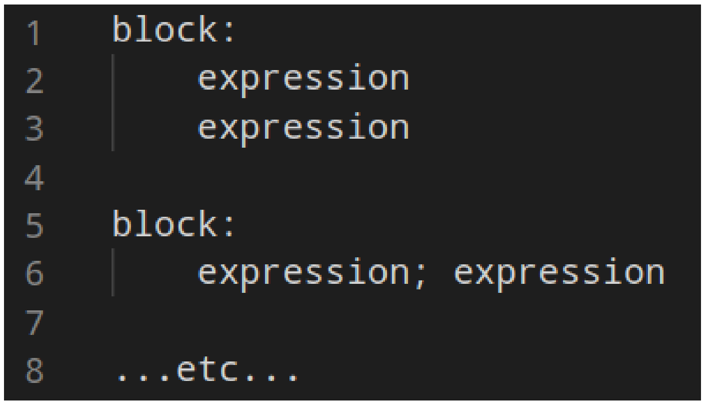
After that, users can simply access the allocated `struct` and easily modify its fields, without having to worry about memory issues. Furthermore, *Omni* provides a way to dynamically allocate multiple instances of a single `struct`, thanks to the in-built `Data` construct. This supports any *Omni* valid type, and it allows the user, for instance, to specify a `param` that will set the number of the entries of the `Data` at `init` time. Furthermore, if the `Data` entries are not explicitly initialized by the user, they will be declared using the default constructor of the specific `struct`. This is a very powerful feature that effectively permits users to create *multiple instances* of algorithms with a single line of `Data` code. For example, if one has developed a `struct Phasor` and wants to allocate 10 of them, it would be as easy as this line of code:

```
phasors = Data[Phasors](10)
```

However, how is all the allocated memory freed? *Omni* implements its own *automatic memory management* scheme that, at its core, simply logs a pointer to every allocated `struct`, and frees it accordingly when the instance of the *Omni* object is being deleted. This last stage depends on how the *wrapper* calls the `Omni_UGenFree` function.

3.5 - Syntax

In this section I will describe all the *blocks* that compose the *Omni* language on a syntactic level. A *block* is defined by a keyword followed by a colon. The body of the *block* needs to be indented, delimiting its scope. Each *expression*, then, happens on a single line of code. Optionally, *expressions* can be separated by a semicolon, allowing them to co-exist on the same line.



```
block:
    expression
    expression

block:
    expression; expression

...etc...
```

Figure 16: *Block-based* syntax in *Omni.*

All *Omni* blocks have access to the global constants `samplerate`, `bufsize`, `pi` and `twopi`.

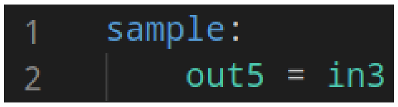
3.5.1 - IO: *ins*, *params*, *buffers* and *outs*

The `ins`, `outs`, `params` and `buffers` blocks allow *Omni* to interface with the external world. The first two permit users to define the number of inputs and outputs of the algorithm. The last ones are used to define specific modes of interaction with the algorithm, as it will be later explained. They all provide a similar syntax, while their scope is different. This *family of blocks* share some common properties:

1 - Dynamic access. In code, specific entries can be accessed via an array syntax: `ins[i]`, `outs[i]`, `params[i]`, `buffers[i]`.

2 - Optional declaration. All of *IO* blocks are optional, which means that if they are not declared, they are simply defaulted to 0. A special case is handled for `ins` and `outs`, which implement a dynamic counting behaviour in the `perform` / `sample` blocks. In fact, `ins` and `outs` do not necessarily require to be declared, but they can be inferred by the highest `in...` / `out...` variable used in the algorithm. For example, the following example would

be a valid *Omni* code, and it will be inferred that the `ins` should have a value of 3 (`in3` is the highest), and `outs` should have a value of 5 (`out5` is the highest):

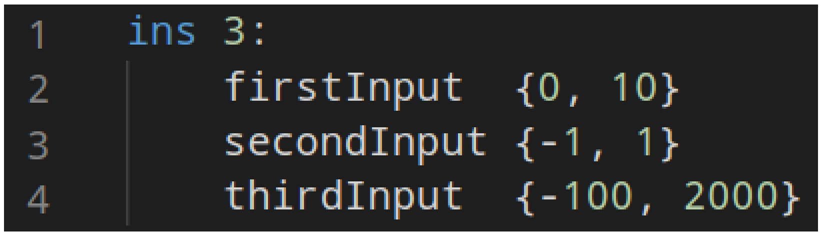


```
sample:
    out5 = in3
```

Figure 17: Dynamic count for *ins* and *outs.*

3.5.1.1 - The *ins* block

The `ins` block defines the number of audio inputs that the algorithm requires. A basic `ins` definition can be as simple as: `ins 3`. This will declare three variables, `in1`, `in2` and `in3` that will reference the relative audio inputs. These will only be accessible in the `perform` and `sample` blocks, the ones that specify the *sample-by-sample* behaviour of the code, as it will be presented later. Optionally, the `ins` block allows users to specify custom names and ranges for the inputs.



```
ins 3:
    firstInput  {0, 10}
    secondInput {-1, 1}
    thirdInput  {-100, 2000}
```

Figure 18: Usage of the *ins* block to declare 3 audio inputs with specific names and minimum / maximum ranges.

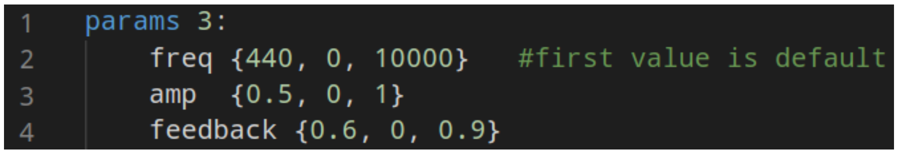
This example shows the definition of three inputs, named `firstInput`, `secondInput` and `thirdInput`. Their values will be automatically clipped by the range expressed in the curly brackets. This feature allows to easily define minimum and maximum values, with no need to explicitly use other constructs such as `if` statements in the code. Moreover, the expressed input names will be available as *aliases* to `in1`, `in2` and `in3`.

3.5.1.2 - The *outs* block

Similarly to the `ins` block, the `outs` block defines the audio outputs of the algorithm. Unlike `ins`, named `outs` are not used in the *Omni* code, as they could create confusion with the declaration of normal variables. For example, if an `out` has the name `firstOutput`, the assignment in the `sample` block '`firstOutput = someValue`' would have the same semantic value as a variable assignment. This problem does not arise for `ins`, as they are *constants* that are not assignable. Perhaps, if a custom syntax will be designed to assign outputs, like *SOUL*'s `<<` operator, this issue could be erased. However, the names are still used for the exported metadata when the `--exportIO` compiler flag is enabled.

3.5.1.3 - The *params* block

The `params` block allows to specify a number of *control rate parameters* to be used in the algorithm. *Control rate* means that their value is not updated on a *sample-by-sample* level, but once per audio block. `params` are available in the `init` and `perform` / `sample` blocks. The syntax is highly similar to the `ins` one. The only addition is the possibility of defining a *default* value for a parameter together with the *minimum* and *maximum* ranges.

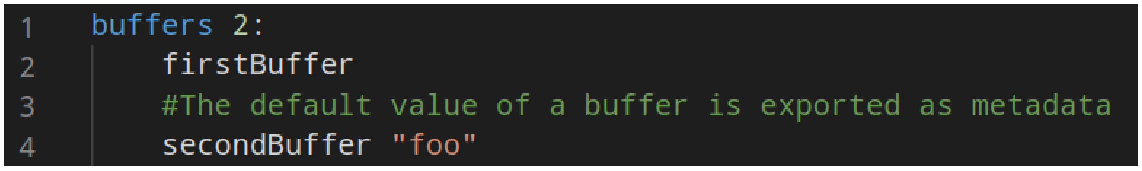


```
params 3:
    freq {440, 0, 10000}   #first value is default
    amp  {0.5, 0, 1}
    feedback {0.6, 0, 0.9}
```

Figure 19: Usage of the *params* block.

3.5.1.4 - The *buffers* block

The `buffers` block is used to specify external memory that is being used in the *Omni* algorithm. The way this memory is handled is defined on a per-wrapper basis with very simple requirements, which will be later shown in the *3.6* section. For end users, no work has to be done: they simply have to declare a `buffers` block to use memory in the preferred *creative coding environment* or wrapper. An in-depth explanation on how to deal with `buffers` in the actual code will be provided later on in the context of explaining `structs`.

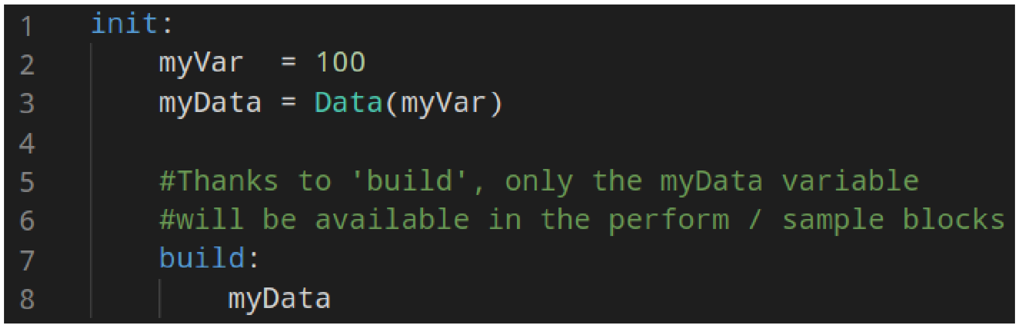


```
buffers 2:
    firstBuffer
    #The default value of a buffer is exported as metadata
    secondBuffer "foo"
```

Figure 20: Usage of the *buffers* block. The default values are exported as metadata.

3.5.2 - The *init* block

The `init` block is a fundamental element in *Omni* as it allows users to define the execution of a specific action to initialize the state of the algorithm. Here it is possible to create `structs` and declare variables that will be used later in the `perform` / `sample` blocks. All variables declared in the `init` block are automatically passed to the successive `perform` / `sample` blocks, unless a `build` construct is defined. The `build` block only allows certain elements to be passed through. By default, every variable declared in the `init` block will be made available in the `perform` / `sample` blocks. With `build`, only the specified ones are passed through. Finally, the `init` block is optional, in case the user does not need to declare re-usable variables or `structs`.

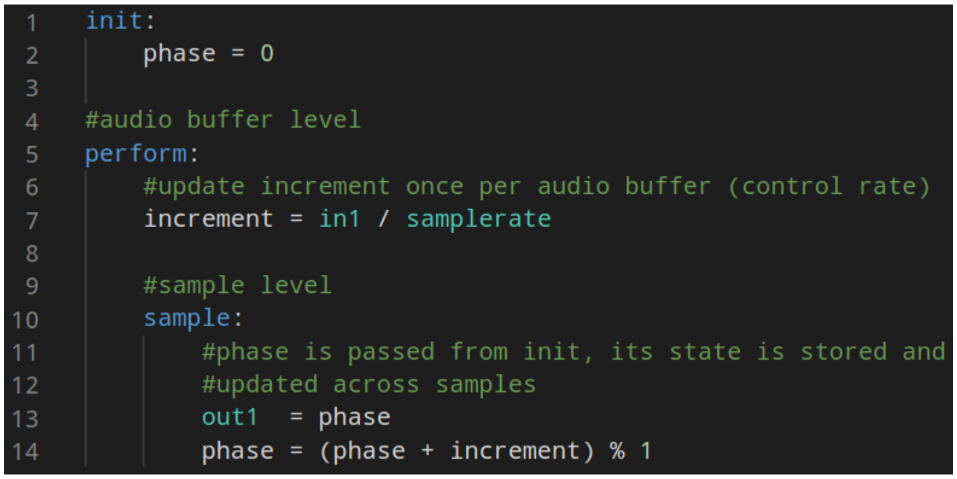


```
init:
    myVar   = 100
    myData = Data(myVar)

    #Thanks to 'build', only the myData variable
    #will be available in the perform / sample blocks
    build:
        myData
```

Figure 21: Usage of the *init* block.

3.5.3 - Core of an audio algorithm: *perform* and *sample*

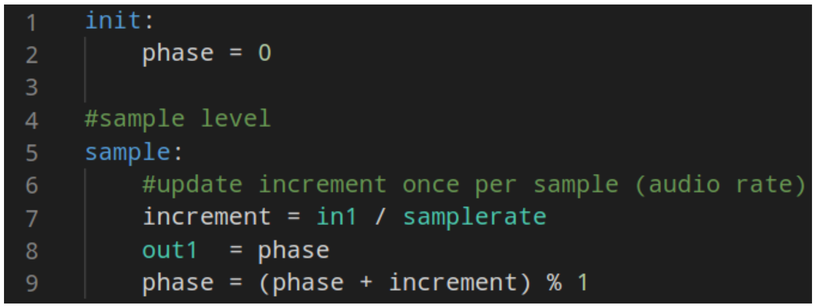
The `perform` and `sample` blocks represent the core of the audio algorithm, allowing the coder to define the *sample-by-sample* behaviour.



```
init:
    phase = 0

#audio buffer level
perform:
    #update increment once per audio buffer (control rate)
    increment = in1 / samplerate

    #sample level
    sample:
        #phase is passed from init, its state is stored and
        #updated across samples
        out1  = phase
        phase = (phase + increment) % 1
```

Figure 22: Usage of the *perform* / *sample* blocks for a simple phasor.

`perform` specifies the *audio buffer* level, while `sample` delimits the *sample* level. In other terms, the `perform` block will be executed once per *audio buffer*, while the `sample` block will be called each audio sample. It is relevant to underline that the `sample` block can be declared on its own, without being included in a `perform` block. Usually, one would want to declare a `perform` block when there is the need of updating certain values only once per audio buffer, instead of once per sample. Depending on the case, this can speed up the performance of certain algorithms. For example, one could rewrite the previous code with only using the `sample` block, in which case the `increment` is performed *sample-by-sample*.



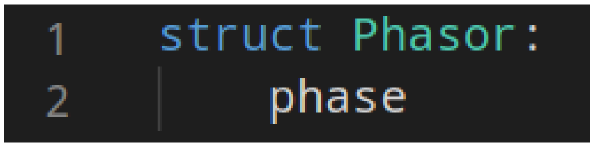
```
init:
    phase = 0

#sample level
sample:
    #update increment once per sample (audio rate)
    increment = in1 / samplerate
    out1  = phase
    phase = (phase + increment) % 1
```

Figure 23: Same algorithm, but using only the *sample* block.

As shown in the code, a fundamental property of these blocks is the inheritance of all the variables declared in the `init` block and optionally forwarded via the `build` construct. This allows those variables to have their state updated during the execution of the algorithm. The `perform` and `sample` are the only mandatory blocks for *Omni* code to compile successfully.

3.5.4 Defining objects with *struct*

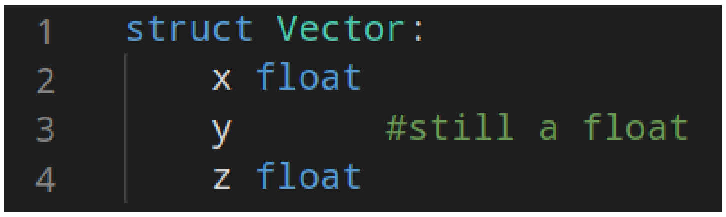
The `struct` construct allows users to define custom objects that can be used in developing specialized algorithms.



```
struct Phasor:
    phase
```

Figure 24: A very basic *struct*, only storing a *float* entry named *phase.*

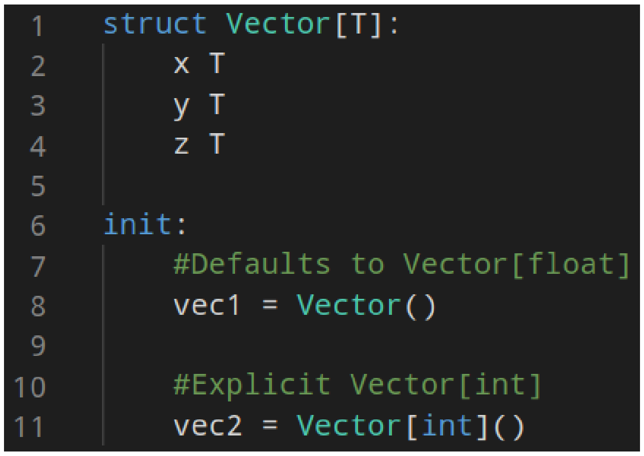
The syntax is very simple: the list of *fields* that the `struct` is composed of is declared after the definition of the name of the `struct`. Following the p*ythonic* style syntax, the fields are indented after the definition of the block via the colon. The types of the fields can be explicitly expressed. If omitted, they're defaulted to `float`.



```
struct Vector:
    x float
    y       #still a float
    z float
```

Figure 25: Another basic *struct* storing three floats.

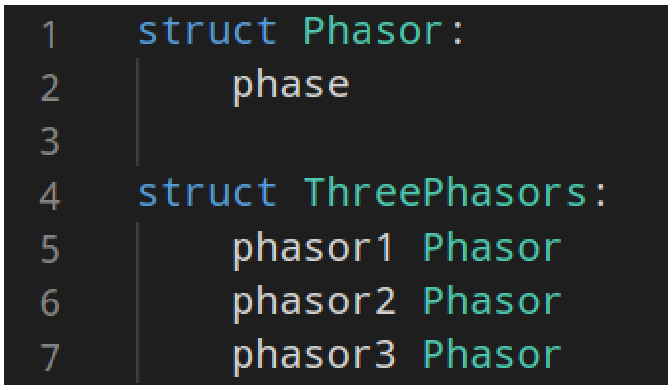
Additionally, `structs` support the use of generics. Generics in *Omni* only work with numeral types.



```
struct Vector[T]:
    x T
    y T
    z T

init:
    #Defaults to Vector[float]
    vec1 = Vector()

    #Explicit Vector[int]
    vec2 = Vector[int]()
```

Figure 26: Declaring a generic *struct* and instantiating two entries in the *init* block.

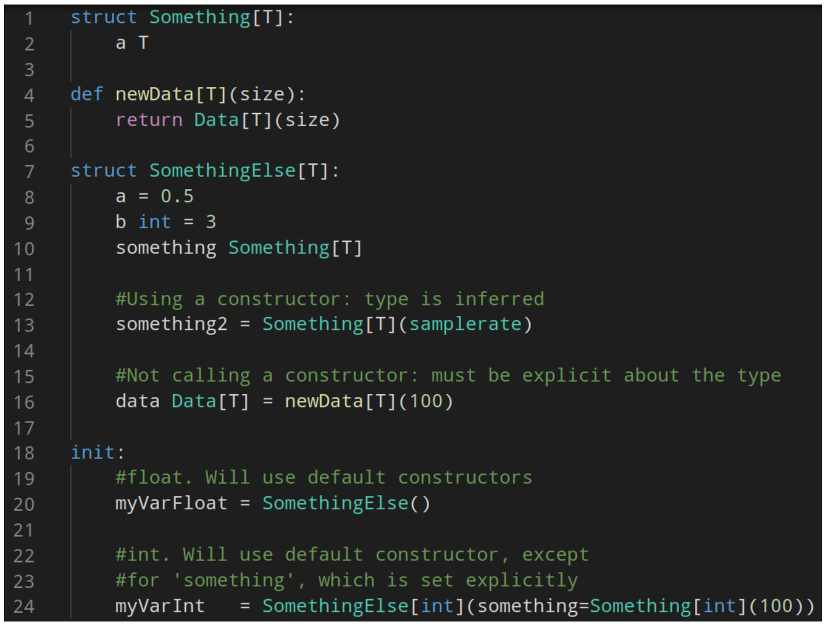
Furthermore, `structs` can contain other `structs`:



```
struct Phasor:
    phase

struct ThreePhasors:
    phasor1 Phasor
    phasor2 Phasor
    phasor3 Phasor
```

Figure 27: *struct* containing *structs.*

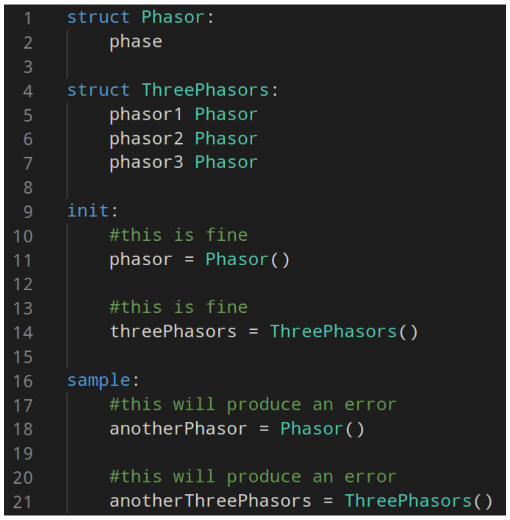
Finally, `structs` support a default initialization value for each field. This value can either be a `struct` constructor, in which case *Omni* will infer the type, or any `def`, as long as the type of the field is explicit. `defs` are *Omni*'s functions, as explained later.



```
struct Something[T]:
    a T

def newData[T](size):
    return Data[T](size)

struct SomethingElse[T]:
    a = 0.5
    b int = 3
    something Something[T]

    #Using a constructor: type is inferred
    something2 = Something[T](samplerate)

    #Not calling a constructor: must be explicit about the type
    data Data[T] = newData[T](100)

init:
    #float. Will use default constructors
    myVarFloat = SomethingElse()

    #int. Will use default constructor, except
    #for 'something', which is set explicitly
    myVarInt   = SomethingElse[int](something=Something[int](100))
```

Figure 28: Complex *struct* with default values.

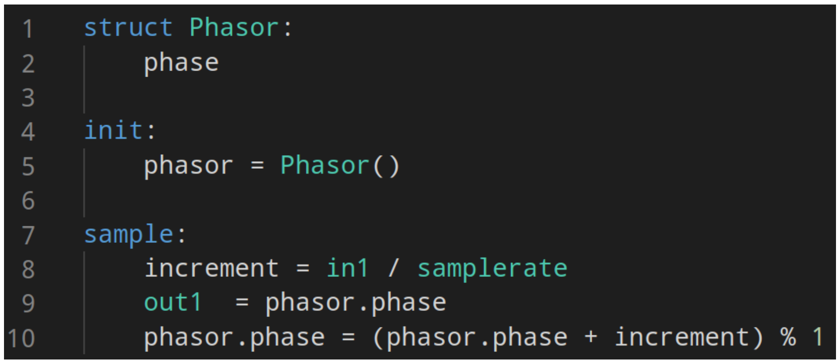
`structs` can only be created in the context of the `init` block. If created in the `perform` or `sample` *blocks*, a compile-time error will be triggered.



```
struct Phasor:
    phase

struct ThreePhasors:
    phasor1 Phasor
    phasor2 Phasor
    phasor3 Phasor

init:
    #this is fine
    phasor = Phasor()

    #this is fine
    threePhasors = ThreePhasors()

sample:
    #this will produce an error
    anotherPhasor = Phasor()

    #this will produce an error
    anotherThreePhasors = ThreePhasors()
```

Figure 29: *structs* can only be instantiated in the *init* block.

Using the `struct Phasor` definition, the previous example code in the *3.5.3* section can be re-written as:



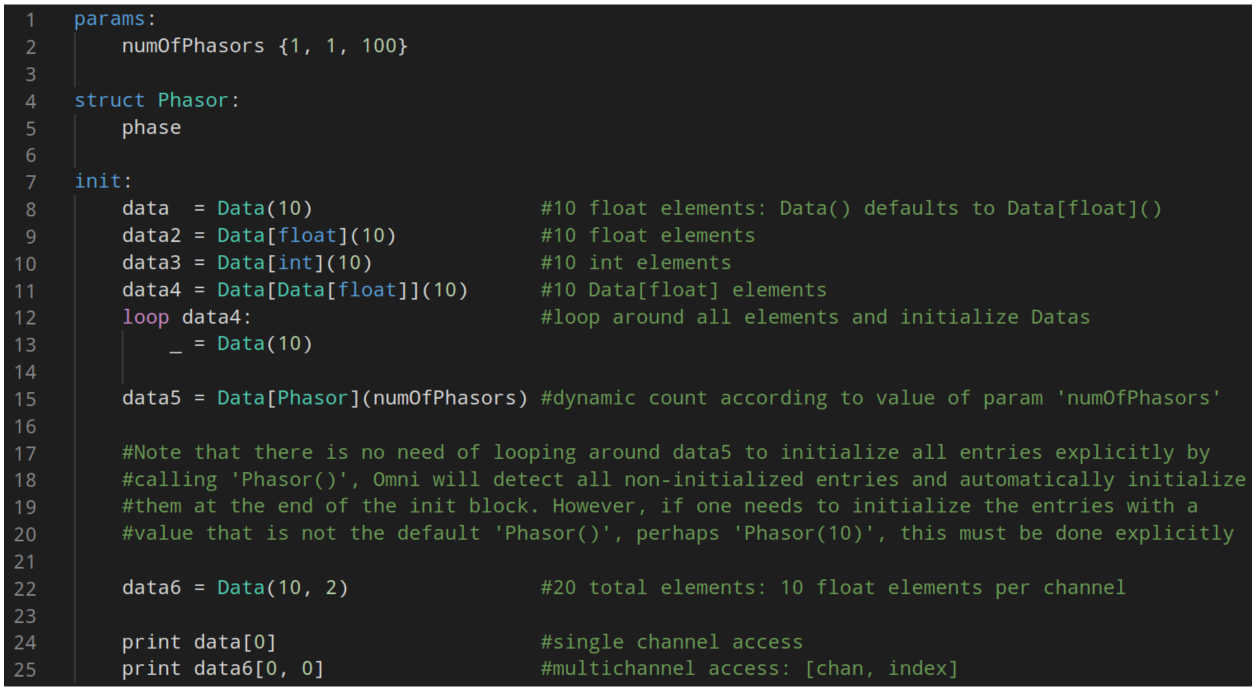
```
struct Phasor:
    phase

init:
    phasor = Phasor()

sample:
    increment = in1 / samplerate
    out1   = phasor.phase
    phasor.phase = (phasor.phase + increment) % 1
```

Figure 30: Using a *struct* to hold the phase value of the phasor.

3.5.4.1 In-built *structs*: *Data*, *Delay* and *Buffer*

Omni offers some in-built `structs` to serve different purposes: `Data`, `Delay` and `Buffer`. The first one represents *Omni*'s way of allocating memory of any kind. `Data` supports every

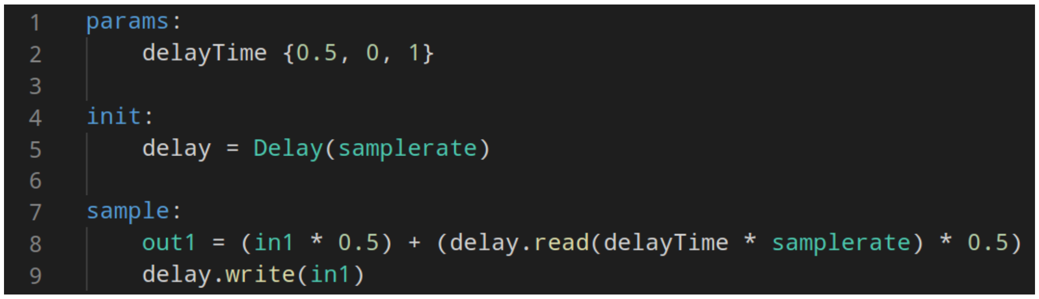
Omni type, including custom `structs`. Furthermore, for all the non-explicitly initialized entries, `Data` will automatically call the default constructor of the specific type at the end of the `init` block. Therefore, once a `Data` of a specific type is declared, it is not often required to call the constructor for each element of the `Data`, unless a specific one is required. *Omni* will detect any non-initialized entries and construct them. This is what happens in the following example of the `data5` variable. Moreover, *Omni* allows users to specify the number of elements of a `Data` at runtime. In fact, whenever a new instance of the compiled object is created, it is possible to retrieve the current value of a specific `param` to determine such behaviour. It is the case of the `data5` variable, which allocates a specific number of `Phasors` according to the value of `numOfPhasors` at initialization time. For instance, in the case of compiling such code with `omnimax`, one could then specify `numOfPhasors` as an attribute when creating the object: `myObject~ @numOfPhasors 50`.



```
params:
    numOfPhasors {1, 1, 100}

struct Phasor:
    phase

init:
    data  = Data(10)                   #10 float elements: Data() defaults to Data[float]()
    data2 = Data[float](10)            #10 float elements
    data3 = Data[int](10)              #10 int elements
    data4 = Data[Data[float]](10)      #10 Data[float] elements
    loop data4:                        #loop around all elements and initialize Datas
        _ = Data(10)

    data5 = Data[Phasor](numOfPhasors) #dynamic count according to value of param 'numOfPhasors'

    #Note that there is no need of looping around data5 to initialize all entries explicitly by
    #calling 'Phasor()', Omni will detect all non-initialized entries and automatically initialize
    #them at the end of the init block. However, if one needs to initialize the entries with a
    #value that is not the default 'Phasor()', perhaps 'Phasor(10)', this must be done explicitly

    data6 = Data(10, 2)                #20 total elements: 10 float elements per channel

    print data[0]                      #single channel access
    print data6[0, 0]                  #multichannel access: [chan, index]
```

Figure 31: Different ways of creating, initializating and accessing a *Data.*

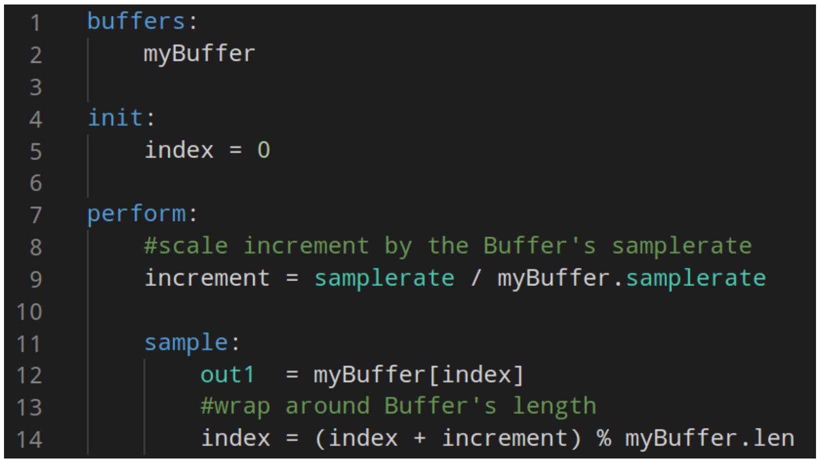
`Delay` is a convenience built-in `struct` to define delay lines. Essentially, it is a `Data` with specific methods to *read* / *write* from the delay line.



```
params:
    delayTime {0.5, 0, 1}

init:
    delay = Delay(samplerate)

sample:
    out1 = (in1 * 0.5) + (delay.read(delayTime * samplerate) * 0.5)
    delay.write(in1)
```

Figure 32: A simple delay line.

`Buffers` are declared with the `buffers` construct. The actual implementation of a `Buffer` is dependent on the *wrapper* that is being used to compile the current *Omni* code; `buffers` only work within an *Omni wrapper*, as illustrated in section *3.5*. For ease of explanation, it can be assumed that a `Buffer` is simply an externally allocated `Data[float]`.

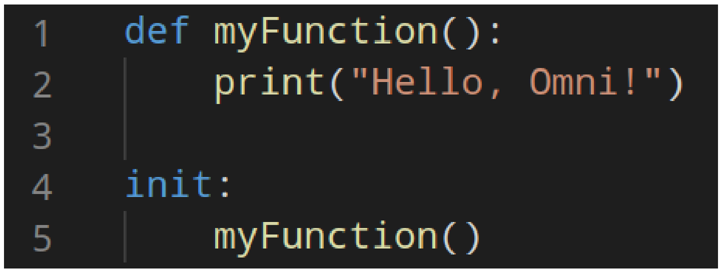


```
buffers:
    myBuffer

init:
    index = 0

perform:
    #scale increment by the Buffer's samplerate
    increment = samplerate / myBuffer.samplerate

    sample:
        out1  = myBuffer[index]
        #wrap around Buffer's length
        index = (index + increment) % myBuffer.len
```

Figure 33: Reading from a *Buffer.*

3.5.6 *def*: *Omni*'s functions

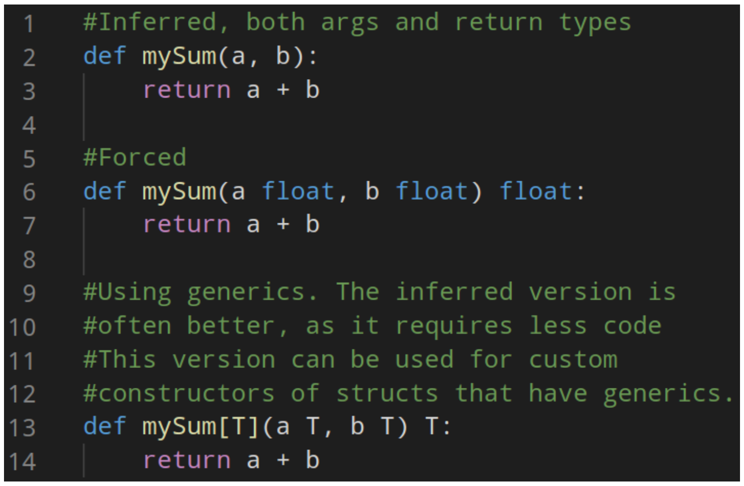
Just as any other programming languages, it is often more convenient to wrap specific portions of code into re-usable functions. *Omni*'s functions are called `def`, like *Python*'s. A `def` introduces a custom scope, but it has access to the `samplerate`, `bufsize`, `pi` and `twopi` global variables. A simple `def` looks like this:



```
def myFunction():
    print("Hello, Omni!")

init:
    myFunction()
```

Figure 34: Basic usage of *def.*

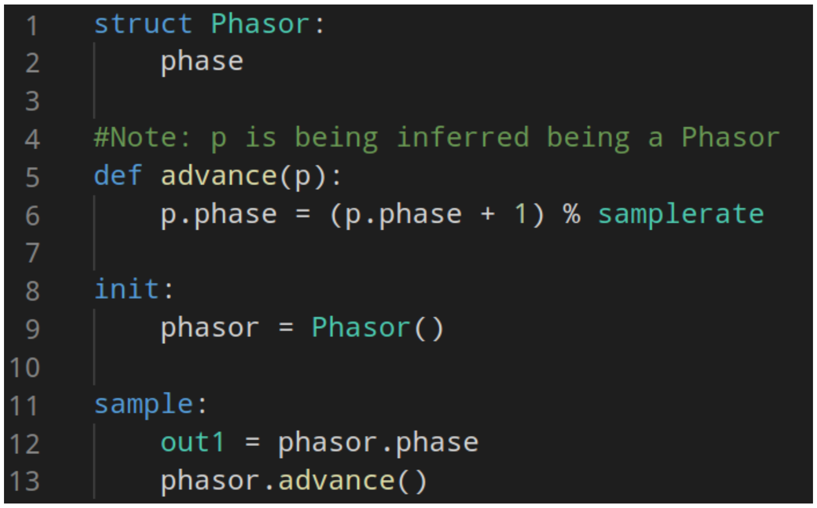
A `def` can have arguments, and the types of these can be inferred by *Omni*. It is possible to force the type of the arguments and of the return type. Furthermore, similarly to `structs`, `def` supports generics. Generics only support number types, and, in the case of `defs`, are mostly used to define custom constructors for `structs` that have generics. For standard usage, in fact, type inference is a better approach to specify a function to perform on multiple types.



```
#Inferred, both args and return types
def mySum(a, b):
    return a + b

#Forced
def mySum(a float, b float) float:
    return a + b

#Using generics. The inferred version is
#often better, as it requires less code
#This version can be used for custom
#constructors of structs that have generics.
def mySum[T](a T, b T) T:
    return a + b
```

Figure 35: Different ways of declaring a *def.*

`defs`, thanks to the dot calling syntax, can be used to implement specific functions for `structs`, resembling of *class methods*. This feature is what makes *Omni* extremely modular: developers can develop their own libraries of `structs` and `defs` that can then be imported (see section *3.4.6*) and used across different projects.



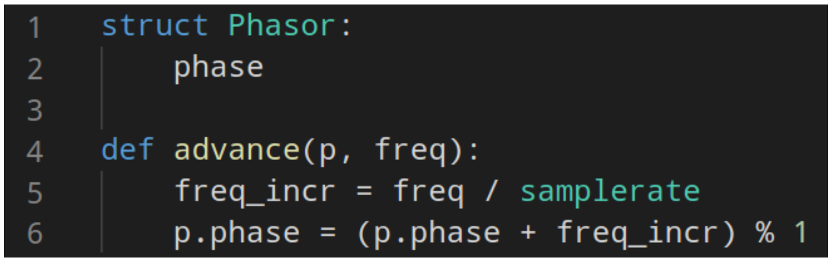
```
struct Phasor:
    phase

#Note: p is being inferred being a Phasor
def advance(p):
    p.phase = (p.phase + 1) % samplerate

init:
    phasor = Phasor()

sample:
    out1 = phasor.phase
    phasor.advance()
```

Figure 36: Calling a *struct* / *def* pair with the method syntax.

3.5.7 Code composition via the *use* block

One of *Omni*'s main features is the ease of code abstraction and modularity. Implementations of specific algorithms can be developed using `structs` and `defs`, and then imported by using the `use` block:

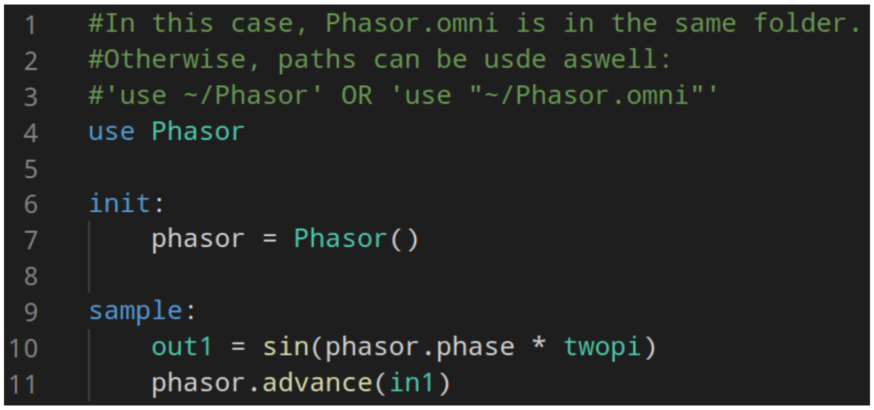
`Phasor.omni`



```
struct Phasor:
    phase

def advance(p, freq):
    freq_incr = freq / samplerate
    p.phase = (p.phase + freq_incr) % 1
```

Figure 37: The '*Phasor.omni'* file.

`Sine.omni`

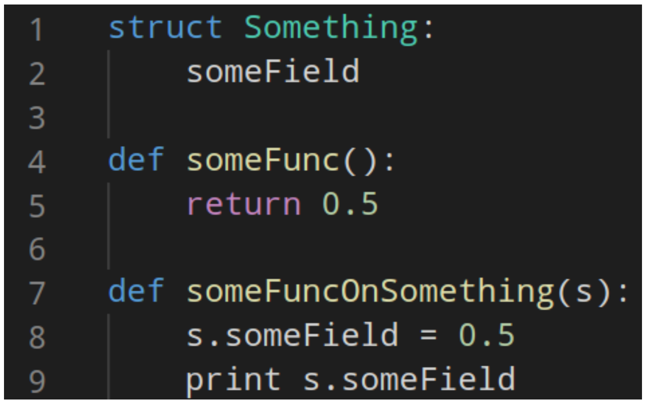


```
#In this case, Phasor.omni is in the same folder.
#Otherwise, paths can be usde aswell:
#'use ~/Phasor' OR 'use "~/Phasor.omni"'
use Phasor

init:
    phasor = Phasor()

sample:
    out1 = sin(phasor.phase * twopi)
    phasor.advance(in1)
```

Figure 38: The '*Sine.omni'* file. This is the file it is going to be compiled

One fundamental feature of the `use` block is its support for defining *aliases* when importing. Thanks to the `as` syntax, it is allowed to import different implementations with the same name from different files without generating any naming collisions:

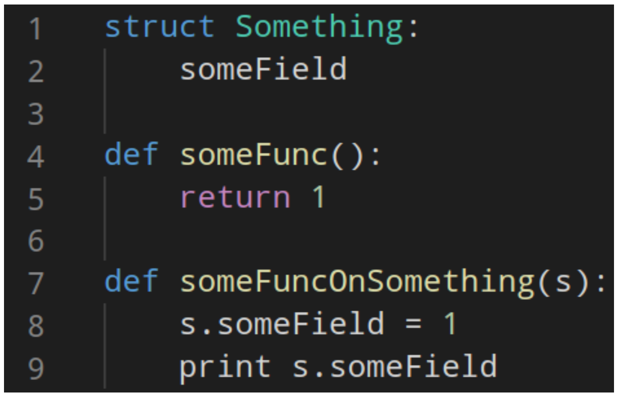
`FirstModule.omni`



```
struct Something:
    someField

def someFunc():
    return 0.5

def someFuncOnSomething(s):
    s.someField = 0.5
    print s.someField
```

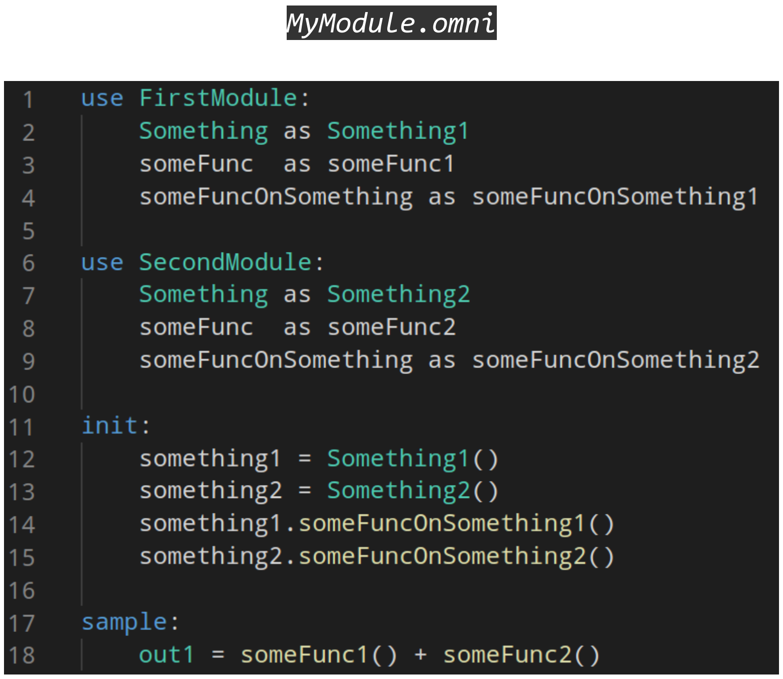
Figure 39: '*FirstModule.omni*'.

`SecondModule.omni`



```
struct Something:
    someField

def someFunc():
    return 1

def someFuncOnSomething(s):
    s.someField = 1
    print s.someField
```

Figure 40: '*SecondModule.omni*'.



```
MyModule.omni

use FirstModule:
    Something as Something1
    someFunc  as someFunc1
    someFuncOnSomething as someFuncOnSomething1

use SecondModule:
    Something as Something2
    someFunc  as someFunc2
    someFuncOnSomething as someFuncOnSomething2

init:
    something1 = Something1()
    something2 = Something2()
    something1.someFuncOnSomething1()
    something2.someFuncOnSomething2()

sample:
    out1 = someFunc1() + someFunc2()
```

Figure 41: '*MyModule.omni*'. This is the file it is going to be compiled.

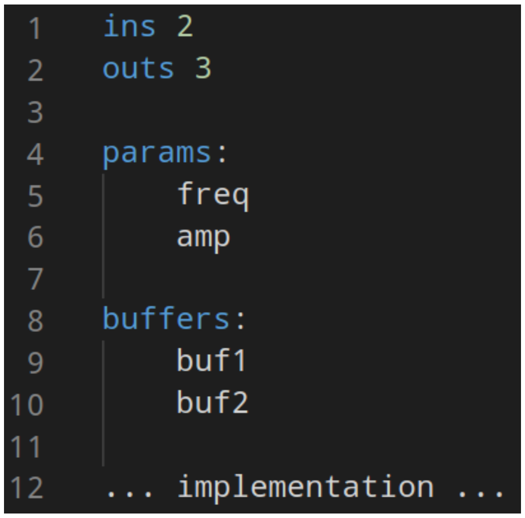
3.6 - Wrappers: *OmniCollider* and *OmniMax*

Omni has been developed from the start as a standalone *DSL* easily embeddable into other projects. I believe that this aspect confers a clear identity on its own to the project, without the need of having a clear dependence on other coding environments. Furthermore, this allows me to keep the development of the *Omni* repository focused on just the core of the language and its features. Being a solo developer, it is crucial for me not to deal with the overhead of maintaining code for each platform/environment that *Omni* would support, but to delegate it to whoever needs to utilize *Omni* in their own projects.

Omni, together with the static or shared library, provides an *'omni.h'* file for easy interoperability with the compiled code. As a consequence, any language that can load shared libraries can quickly utilize *Omni*'s binaries. Additionally, *Omni* provides an even deeper connection by providing a simple *API* to define *wrappers* that operate at the code generation level, specifically to support the `Buffer` interface for the specific environment. This is certainly not necessary if the desired wrapper will not use `Buffers`, but code that contains the `buffers` block will not compile, as it does not for the standard `omni` command unless a valid `--wrapper` is provided.

I will here present two examples of wrappers that I wrote for *Max* and *SuperCollider*, respectively called *OmniMax* and *OmniCollider*. These provide an easy one-step compilation of *Omni* code for creative coders. All the process related to getting the specific project's *SDK* and *header files* and setting up the *C++* sources that call into the *Omni* compiled binaries are abstracted away by simply using the `nimble` installer: `nimble install omnimax` / `nimble install omnicollider` will take care of all needed operations, including the installation of the correct `omni` compiler. Under the hood, both *OmniMax* and *OmniCollider* follow the same structure, which can perhaps serve as a starting point for others to develop their own wrappers. At their core, they both simply use the `omniBufferInterface` *Nim* macro to specify the correct implementation to access `Buffers` for the individual platform. The rest of the code simply maps the environment's specific calls into *Omni*'s exported *C API*, e.g. to deal with `params`.

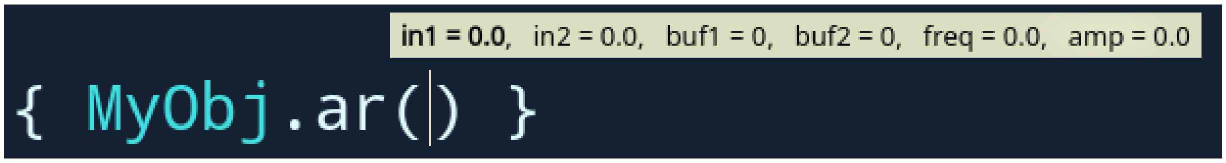
To show how an *OmniCollider* and *OmniMax* object works within the specific environment, let us consider the four blocks that make up how *Omni* communicates with the external world: `ins`, `outs`, `params` and `buffers`.



```
ins 2
outs 3

params:
    freq
    amp

buffers:
    buf1
    buf2

... implementation ...
```

Figure 42: IO in '*MyObj.omni'.*

In the case of OmniCollider, the compiled *UGen* will treat `ins` as audio rate arguments, while `params` and `buffers` will be considered control rate ones. These will also be converted to the right rate if the user fails to do so. In the case of multiple `outs`, *OmniCollider* will declare the *UGen* as a *MultiOutUGen*, as per *SuperCollider*'s specification.



```
in1 = 0.0,  in2 = 0.0,  buf1 = 0,  buf2 = 0,  freq = 0.0,  amp = 0.0
{ MyObj.ar() }
```

Figure 43: *SuperCollider* interface for the IO of an *Omni* compiled *UGen.*

In the case of *OmniMax*, the `ins` and `outs` blocks are respectively mapped to the audio inlets and outlets of the object. On the other hand, `params` and `buffers` can be set via messages and attributes. Furthermore, default values can be set directly in the body of the *Max object*: numbers will set `params` and symbols will set `buffers`.

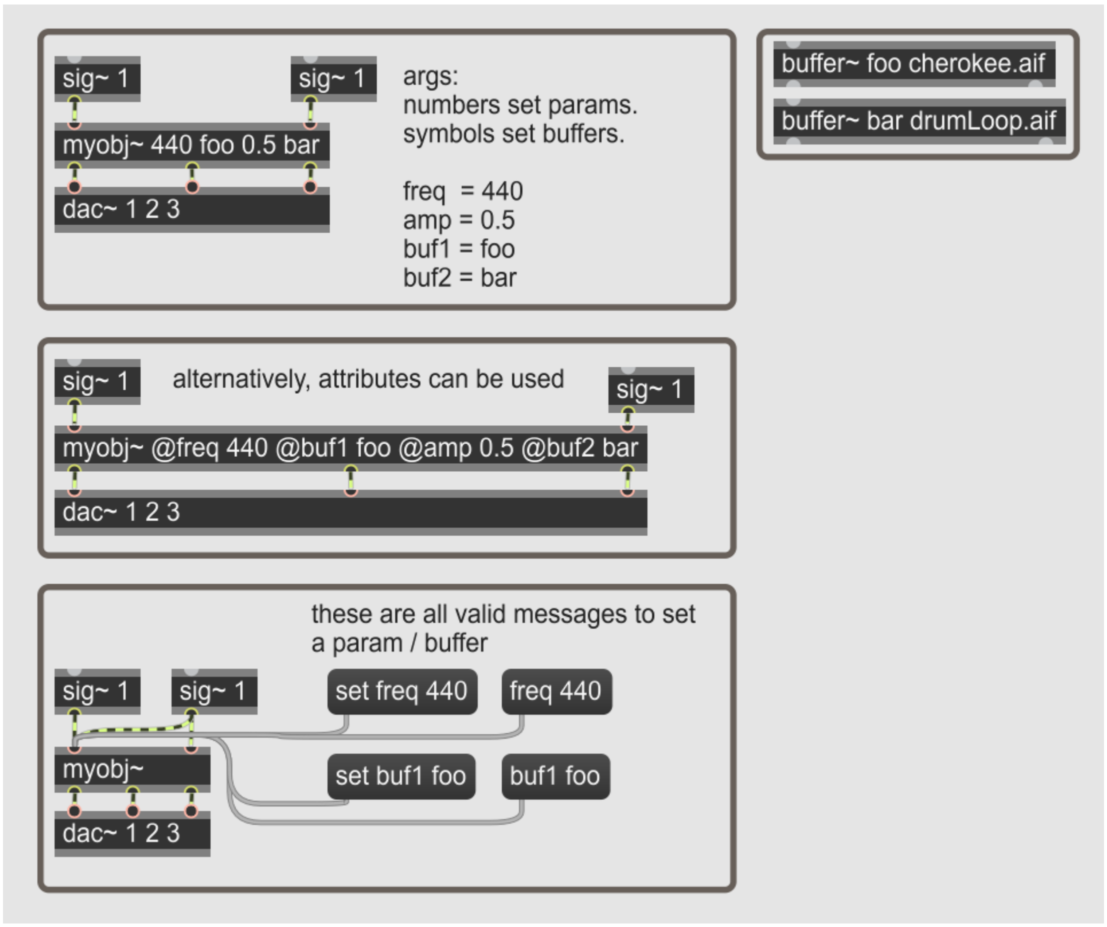


Figure 44: *Max* interface for the IO of an *Omni* compiled *object.*

Chapter 4 - *Alga*: interpolating live coding language

4.1 - What is *Alga*?

Alga is a new language for *live coding*. Its focus is on the interpolation of the connections between different dynamic modules at runtime. Hopefully, this will be clear to the reader by the end of the chapter. At the time of writing this thesis, *Alga* is not in its final *linguistic* form, but it is only comprised of the *AlgaLib* library, which is the core implementation developed in *SuperCollider*. *AlgaLib* already includes the main concepts of *Alga*, and it works well within the *sclang* syntax. However, in the future, this implementation will only be called from a *custom language* that will be using *AlgaLib* and *SuperCollider* as a backend, similarly to the way in which *TidalCycles* works with its library *SuperDirt*. For reasons of clarity, when I refer to *Alga* in this thesis, I am solely talking about the *AlgaLib* implementation.

Like the choice of *Nim* for *Omni*, working with *SuperCollider* allowed me not to have to deal with building a multi-platform audio application to support my work. Instead, it permitted me to only focus on the novelties that I wanted to implement with *Alga*. The dynamism of a language like *sclang*, in conjunction with the powerful *scsynth* and *supernova* synthesis engines, proved to be the perfect choice for this specific project, as they already included a sparse library of *Unit Generators* inside an already proven ecosystem. Finally, I believe that the choice of developing parts of *Alga* in a widespread creative coding environment like *SuperCollider* will hopefully help with the adoption of the project by other people.

4.2 - Contextualizing *Alga*

Alga has been developed in a very flourishing period for *live coding* languages. Over the last ten years, *live coding* has been a very successful practice in the music landscape, both in academic and non-academic circles. The *TOPLAP* community is arguably the most notable organization in the field of *live coding*, taking care of planning shows, workshops and events all around the world (TOPLAP, 2004). From the musicians' perspective, the appeal towards *live coding* languages comes from their quickness and interactivity in developing a sonic/visual live performance via code, often projected on a screen behind the performer. This allows the audience to *participate* and *interpret* the execution of the musician not only from the sonic result, but also through visual stimuli (Roberts, 2016 and Burland & McLean, 2016).

The current main trend of *live coding* languages, and consequently performances, is towards a *looping* approach to music and sound generation: popular languages like *ixi-lang* and *TydalCycles* are designed towards the execution of styles that are heavily *rhythm* or *pattern-based* (Magnusson & McLean, 2018). Personally, while I embrace the *idea* of *live coding* performances and what they represent, I do not engage in the interface that current languages for *live coding* propose. It is surely possible to work around the interface of such tools in order to make them more akin to my idea of music, yet it would still mean to be limited by certain aspects of the languages that are *immovable*. As a composer and performer of long-form noise music, how could I run smooth transitions between different *synthesis nodes* using a pattern language like *TidalCycles* or *FoxDot* (Kirkbride, 2015), whose changes to running *synths* are only applied at the tick of a pattern? I could design such transitions in the definition of a *Synth* itself – the *SynthDef* code in *SuperCollider* terms – but this workaround would only be effective for the specific *instance* of a *Synth*, and it would not be available on a higher level, directly in the language. Taking a non-looping approach to *live coding* does not mean to abandon precise timing or pattern control, as it will be later outlined in the context of the *AlgaScheduler*. In opposition to popular languages *Alga* shifts the attention from the individual values of a pattern to the state of a running *audio module*, similarly to modular synthesizers. As a consequence, *Alga* promotes a style of improvisation that highlights changes that happen *fluently* over windows of time, instead of *instantaneously* on pattern execution.

In the context of *SuperCollider*, why have I not chosen to utilize an already established library like *JITLib* (Rohrhuber & de Campo, 2011), which already allows users to create modules on the fly and connect them together? Firstly, because when establishing a new connection *JITLib* does not interpolate the *parameter* content, but it simply replaces the receiver node with a new one, fading it in with the new connection in place. Fading in and out, *crossfading*, is a completely different process than effectively interpolating the old parameter connection with the new one at the input of the receiver. *JITLib*'s individual elements, called *NodeProxies* and *Ndefs*, utilize *SuperCollider Busses* to determine connections among the different nodes. A *Bus* is a simple audio buffer that can be filled by any running instance of a *Synth* on *SuperCollider*'s audio server. *Alga* takes the same approach, but it expands on it by allowing *AlgaNodes*, which are *Alga*'s elements, to store metadata information about all the connections that are in place between any *AlgaNode* that is patched to the current one. Contrairly to other cl*asses* in *SuperCollider*, *AlgaNodes* are context aware of the entire chain of connections in which they exist, an aspect that provides different benefits. Primarily, it allows *Alga* to automatically re-order groups on the audio server so that there is never a block-size delay between connections, unless feedback is involved. This is not dealt with by *SuperCollider* itself, and it can generate problems where a node might be reading from a *Bus* that has lastly been written to on the previous audio buffer, and not the current one. Such an approach generates a delay that is problematic for situations that need sample-syncing precision. *Alga* prevents this behaviour entirely without requiring the users to change their code. On the other hand, *JITLib* does not take care of group ordering, allowing delays to occur across nodes. Finally, one more benefit is that *Alga* allows any rate and channels combination to be patched anywhere, running appropriate audio to control rate conversions (and vice-versa) and channels re-mappings. These are a fundamental features that permit users to ‘patch anything into anything’, with no need to worry about low-level issues like channels count and rates conversion during a performance. Once more, these are not aspects that will be taken care of for the user by *JITLib*, adding one more overhead for the user to figure out *which* node can be connected to *what*, depending on the correct number of channels and rate.

4.3 - Dynamic patching through interpolation: demystifying mistakes

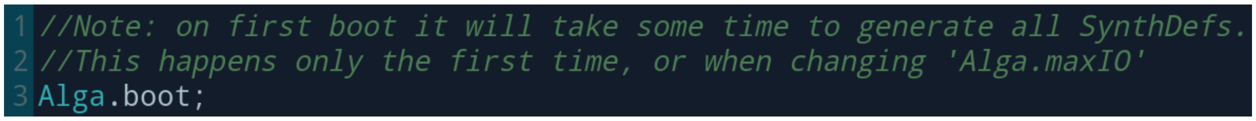
The widespread approach to *live coding* collects a number of features that are mostly useful in the context of producing certain types of *looped* music. Patterns are the core of the implementation, but there surely are sonic structures that do not play well with this notion. In fact, *live coded pattern-based music* arguably prevents the users from the expandability and discovery that *patching* or *connecting* together smaller building blocks allows: each *instrument* in current *live coding* implementations is often an individual entity that does not communicate with the other ones on a *sonic level*. Instead, they are triggered at specific times according to a common *clocking system*. The *modulation* of these instruments, then, does not follow a modular paradigm, where it is derived from the *output* value of another module, but just from the *static* value that the pattern holds at the specific moment in time in which the event is triggered, however quickly this might occur. To explain this, let us consider the case of *TidalCycles*. Any *player*, named from *d1* to *d16* (TidalCycles, 2021), has access to the *SuperDirt* environment to *play* specific *Synths* at a specific time. The arguments to these synths, which set the specific values of certain parameters, are only set *once* at the triggering of the *Synth* instance. This means that their values do not change over the execution of the *Synth*, but remain static throughout. Regardless of the speed of triggering new *Synth* instances, then, the actual content of the parameters is always static. In *TidalCycles*, *FoxDot* or *ixi-lang* it is not possible to connect the *output* of a *Synth* to the *parameter input* of another. Each *player* has its own *modulation scheme*, although it remains confined in itself.

Creative coders, improvisers and composers that do not perform *looped* music usually have to build their own systems either around the limitations of the *out-of-the-box* experience of certain environments in order to fit their specific needs. One such case is the work of Sam Pluta, who designed a modular software environment within *SuperCollider* that he uses for his *computer music* improvisations and compositions (Pluta, 2012). However, being so specifically tailored to a personal practice often makes some of these tools not applicable to other people's concerns. While this view could arguably be applied to my design of *Alga*, I believe that the modular approach to sound making that *Alga* proposes is a well-established paradigm that should ease in users of any expertise. *Alga*'s novelty is surely not in the modularization of sonic processes, but in how these would interact with each other over time. In sound synthesis terms, *patching* a module to another can be regarded as a *static action* the result of which can be dynamic. The *act of patching* an LFO module to the frequency input of an oscillator will always generate a *discontinuity* in time between a *before* and an *after* the connection. The *before* and *after* can surely produce a dynamic result, depending on the modulation sources, but the *moment* of connection is of *immobile* nature. At the lowest level, this means that the *audio sample* in which the new connection takes place abruptly switches from reading values from the *old* modulation source to reading the ones from the *new* one, effectively producing an audible *impulse*. *Alga* proposes an approach to *sonic patching* that enhances the nuances of the *connection*, turning it from a *static moment* to a *dynamic* one. Connecting two modules together in *Alga* does not set the specific parameter instantaneously, but it triggers a process of interpolation between the current state of the receiver and the future one, over a specific amount of time. As a result, the *discontinuity* that comes from *static patching* is eradicated in favor of a *fluid continuity*. This single feature opens up several experimental traits for composition and improvisation that would not be possible in other systems where modules are already often patched together or where, as already suggested, dynamism is created by the parallelization of *individual blocks*. On the other hand, movement and fluidity in *Alga* are inherent traits *embedded* in the way the connections within the system itself work, as a fundamental feature. Among the numerous directions that such characteristics can help with composition and improvisation, one that I personally found valuable in my own practice has been what I would refer to as the *demystification of mistakes* by re-iterating the process of *patching*. For the specific kind of music I perform, taking a *slower* approach to signal patching highlighted how it is not quite as relevant to *make mistakes* in connecting nodes the wrong way. Over a 30 seconds long interpolation that can be re-triggered at any stage, the *mistake* in specifying a new connection would be *absorbed* by the signal flow, sonically resulting in just a temporary state of a system in continuous movement.

4.4 - The *SuperCollider* implementation and technical analysis

This section will be focused on the *SuperCollider* implementation of *Alga*, together with some technical issues that I had to address in order to develop such software.

AlgaLib provides a set of *SuperCollider classes* for the execution of *Alga*. The main class is called `Alga`, and it stores general information about the state of the system. Furthermore, the `Alga` *class* manages the initialization of all the elements and the booting of the synthesis server via the `boot` function. This method allows to optionally pass in an instance of `AlgaServerOptions`, which, similarly to `ServerOptions`, determines the features to boot a server with. On a first `Alga.boot` call, the system will take some time to generate all the necessary `AlgaSynthDefs` that allow dynamic connections to happen. This process happens only once, as the definitions are stored in the default `SynthDefs` directory.



```
//Note: on first boot it will take some time to generate all SynthDefs.
//This happens only the first time, or when changing 'Alga.maxIO'
Alga.boot;
```

Figure 45: To boot *Alga*, simply call *Alga.boot.*

4.4.1 - *Alga*'s atom: *AlgaNode*

The `AlgaNode` class is the core of *Alga*, essentially representing the *atomic module* that can be connected to all the other modules or, likewise, receive their connection. Its concept is definitely inspired by modular synthesis, where *inputs* and *outputs* of a module are the interface through which correlations with the system can happen. The implementation of an `AlgaNode`, similarly to a *SuperCollider*'s `NodeProxy`, is described by either a `Function` or a `Symbol` that points to an `AlgaSynthDef`. From this description, the `AlgaNode` infers metadata about its state, like the number of inputs and outputs, their names and their rates (audio or control). These notions are then stored and used to connect an `AlgaNode` to another by directly referring to specific input and output names.

With `SynthDefs`, *SuperCollider* provides users with an interface to encapsulate sonic algorithms into re-usable definitions. While specifying *named inputs* to the `SynthDefs` is an easy task thanks to the *args* syntax and the `NamedContol` class, the description of *named outputs* is not implemented. To define the *outputs* of a running `Synth`, *SuperCollider* implements the `Out` class, which simply writes values to an *audio* `Bus`. While this allows users to code a single `SynthDef` to write to as many `Busses` as they desire, the *description* of these output signals is not provided in a clear way as it is done for inputs. This promotes a hierarchical distinction between inputs and outputs: a `SynthDef` is only described by its inputs, while its outputs can vary depending on which `Busses` they refer to. This feature contrasts the one of modular synthesizers, where a single module is not only described by its inputs and their names, but also by its outputs. Both inputs and outputs are named and outlined in the same panel together. Consequently, in modular synthesizers inputs and outputs assume the same semantic value, and it is the same with *Alga*. This is the scope of the `AlgaSynthDef` class, which allows users to define an `outsMapping` argument to specify the name of each output of the *UGen function graph*. `outsMapping` are expressed as an `Array` of `Symbols` and `Numbers` / `Arrays` pairs, each representing a specific mapping. The `Symbols` can later be addressed to *patch* specific *outputs* of an `AlgaNode` to specific *inputs* of another one. Furthermore, each output channel is represented by default *names* through the symbols `\out1`, `\out2`, etc... Finally, the `AlgaNode.new` method also provides an `outsMapping` argument, in the case of using a `Function` as definition, instead of a `Symbol` referring to an `AlgaSynthDef`. This allows to specify outputs mapping even for temporary *UGen graph functions*. The following example should clarify these notions:

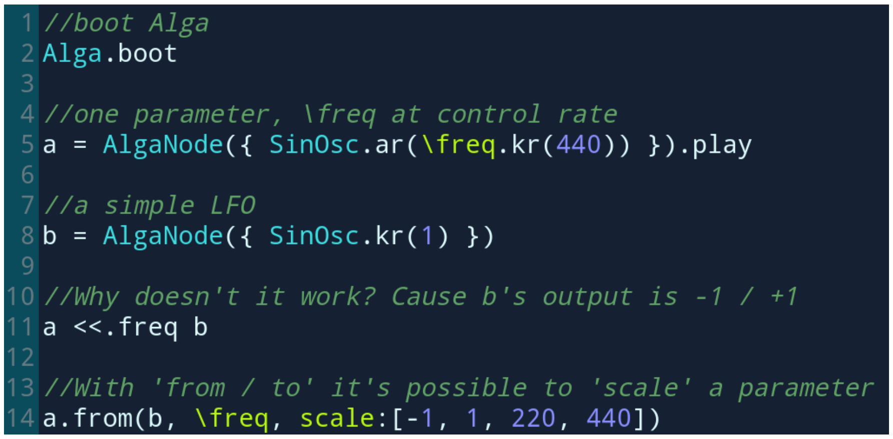
```
//boot Alga
Alga.boot

// \one will have 2 channels,  [0, 1]
// \two will have 2 channels,  [2, 3]
// \three will have 1 channel, [2]
// \four will have 1 channel,  [3]
(
a = AlgaNode(
    { SinOsc.ar([220, 440, 880, 1760])},
    outsMapping: [\one, [0, 1], \two, [2, 3], \three, 2, \four, 3]
)
)

//a simple 2 channel bypass
b = AlgaNode({ \in.ar([0, 0]) }).play

//connects \one == [0, 1]
b.from(a, \in, chans: \one)

//connects \two == [2, 3]. Note that \in is the default param name
b.from(a, chans: \two)
```

Figure 46: Handling of *outsMapping* when making connections.

The previous example introduced the `from` function, which allows an `AlgaNode` to receive the output from another node. Its companion method is called `to`, which reverses the sender with the receiver. Furthermore, the `<<` / `>>` operators are aliases to these functions.

Both paradigms of connecting *from* and *to* an `AlgaNode` are provided because they allow users to express the broader concept of *patching* from the angle that fits their usage more. I personally use them both. In fact, while I prefer to express a modulation connection with the *receiver* syntax – e.g. *receiving* the *LFO* signal *from* a node – I would rather address the *synth to effect* behaviour with the *sender* syntax – e.g. *sending* a signal *to* an effect.

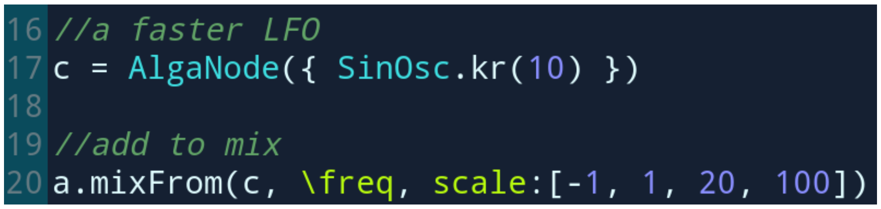
Connecting two modules essentially means mapping some - if not all - of the outputs of one node to a parameter input of another node. If no mapping is explicitly specified, the `AlgaNode` will connect all of its outputs to the specific input parameter of the receiver, making the right conversions of rate and number of channels. Moreover, `AlgaNodes` allow to `scale` a received connection accordingly. For example, this allows users to *connect* the same *LFO* to two different nodes, with the value scaled to fit the specific use case, like controlling the frequency of an oscillator, or the amplitude of a modulation, etc… Lastly, this behaviour helps with defining a library of `AlgaSynthDefs` that, like with the modular synthesizers' CV mapping, have a standardized output range, further incrementing the interoperability of different modules.



```
//boot Alga
Alga.boot

//one parameter, \freq at control rate
a = AlgaNode({ SinOsc.ar(\freq.kr(440)) }).play

//a simple LFO
b = AlgaNode({ SinOsc.kr(1) })

//Why doesn't it work? Cause b's output is -1 / +1
a <<.freq b

//With 'from / to' it's possible to 'scale' a parameter
a.from(b, \freq, scale:[-1, 1, 220, 440])
```

Figure 47: Simple *Alga* example: an *LFO* mapped to an oscillator's frequency parameter.

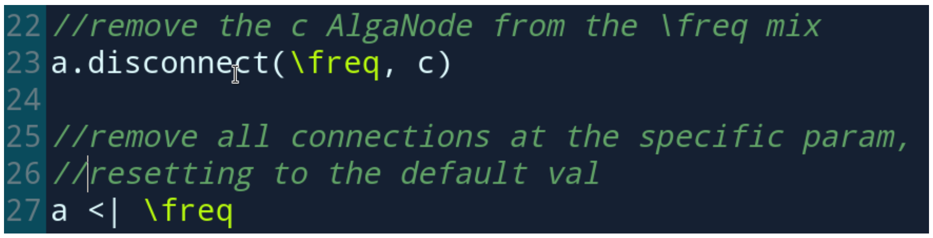
`AlgaNodes` provide the possibility to *mix* the entries at the input, allowing to dynamically add nodes to a specific modulation of a parameter. The functions that enable this behaviour are `mixFrom` / `mixTo`, together with the `<<+` / `>>+` operators. Expanding the previous example, one more *LFO* with a frequency of 10Hz can be added to modulate the same `\freq` parameter.



```
//a faster LFO
c = AlgaNode({ SinOsc.kr(10) })

//add to mix
a.mixFrom(c, \freq, scale:[-1, 1, 20, 100])
```

Figure 48: Adding another *LFO* to modulate the same *\freq* parameter.

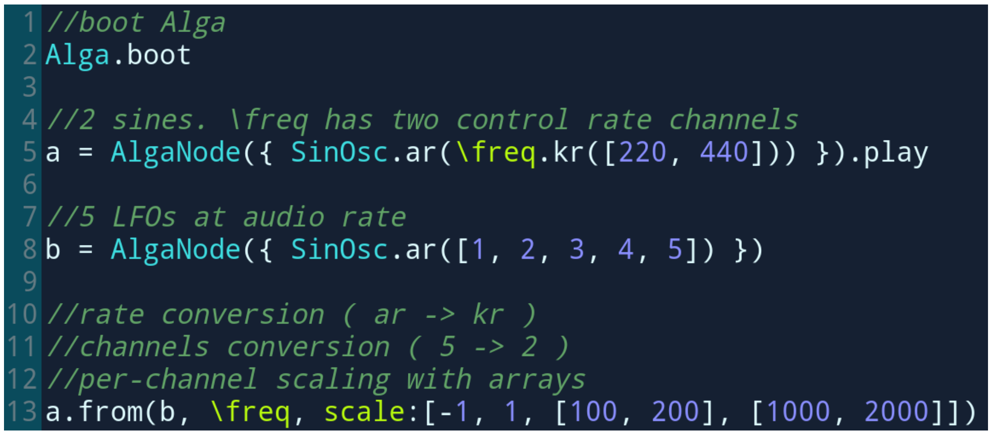
Just as it is possible to *patch* nodes together, it is also possible to remove certain connections with the `disconnect` function or the `<|` operator.



```
//remove the c AlgaNode from the \freq mix
a.disconnect(\freq, c)

//remove all connections at the specific param,
//resetting to the default val
a <| \freq
```

Figure 49: Removing connections.

When making any connection between `AlgaNodes`, *Alga* also takes care of automatically converting the rate and channel count of the sender to match the ones of the receiver. Indeed, it is possible to connect an audio rate `AlgaNode` to a control rate parameter, and vice-versa. Likewise, connecting a 5 channels node to a 2 channels parameter is also permitted by simply reading values from the smallest common number (2) of channels. On the other hand, if the sender has fewer channels than the receiver, *Alga* will wrap the channels of the sender accordingly. Again, this helps with the 'patch anything into anything' approach that *Alga* promotes.



```
//boot Alga
Alga.boot

//2 sines. \freq has two control rate channels
a = AlgaNode({ SinOsc.ar(\freq.kr([220, 440])) }).play

//5 LFOs at audio rate
b = AlgaNode({ SinOsc.ar([1, 2, 3, 4, 5]) })

//rate conversion ( ar -> kr )
//channels conversion ( 5 -> 2 )
//per-channel scaling with arrays
a.from(b, \freq, scale:[-1, 1, [100, 200], [1000, 2000]])
```

Figure 50: Converting audio rate to control rate and mapping channels accordingly.

Finally, *Alga* also ensures that anytime a connection is made between two `AlgaNodes` the respective groups on the *server* are arranged so that the receiver willl always follow the sender. In audio terms, this means that there will not be any delay between the connection, as the process of *writing* and *reading* from the specific `Busses` happen on the same audio buffer. This re-ordering also affects all the connected *nodes*, effectively creating a connection graph that always respects connections order. It is also possible to feedback the nodes' connections, which *Alga* will detect introducing a block-size delay *only* to the feedback connection, not the whole graph.

```
(
//First argument of Alga.boot can be a function to be executed
//as soon as server boots, similarly to Server.waitForBoot
Alga.boot({
    a = AlgaNode({ SinOsc.ar(\freq.kr(50) + \fm.ar(0)) });
    b = AlgaNode({ Saw.ar(\freq.kr(3) + \fm.ar(0)) });
    a.from(b, \fm, scale:[-200, 200]);
    b.from(a, \fm, scale:[-50, 50]); //feedback connection!
    a.play(chans:2); //play stereo
})
)
```
Figure 51: A simple feedback example.

After the introduction of some core ideas about `AlgaNodes`, the main feature of *Alga* can now be illustrated: the interpolation of the connections. Everytime a new connection is established, *Alga* will *interpolate* from the current state of the *patched* parameter to the value of the newly made connection over a specified window of time. *Alga* allows to re-trigger the interpolation at any stage, allowing a dynamic exchange between the coding experiments and the sonic results. In fact, once an interpolation has been triggered, one does not have to wait for the interpolation process to be completed before a new one is specified. Instead, *Alga* will consider the current dynamic state as the new starting point, before starting the movement towards the new one. To enable the interpolation behaviour, every connection function takes a `time` parameter, which sets the length of the transaction. It is also possible to set the interpolation time for all the connections to the `AlgaNode` by setting the `connectionTime` / `ct` variable. Furthermore, the interpolation time can be specifically indicated for a single parameter and all its subsequent *patching* by calling the `connectionTime_` function with the parameter symbol as an extra argument.

```
//Boot Alga
Alga.boot

//3 seconds of interpolation time
a = AlgaNode({ SinOsc.ar(\freq.kr(440)) }, connectionTime:3).play(chans:2)

//Now it will interpolate
a <<.freq 220

//Random linear noise
b = AlgaNode({ LFNoise1.kr(2) })

//if 2 values, scale assumes the range -1 to 1
a.from(b, \freq, scale:[220, 440])

//Disconnect, it will go back to 440, the default value
a <| \freq
```
Figure 52: Core of *Alga*: interpolating connections.

Any definition of an `AlgaNode` is not static once it has been set, but it can be replaced at any time with a call to the `replace` function. In such case, *Alga* will maintain all connections to the node being replaced for all the parameters that keep the same names. Furthermore, *Alga* will re-connect the node to all the *receivers* that it was connected to as a *sender*, respecting the *connectionTimes* of all of them. The `replace` function is an essential feature in developing a dynamic system like *Alga*. By providing the option of not only ‘patching anything into anything’, but also of ‘replacing anything with anything’, *Alga* allows users to model the way in which they use the software at each stage: any *action* is thus rendered reversible with the same behaviour that triggered it in the first place.

```
//Boot Alga
Alga.boot;

//Sine
a = AlgaNode({ SinOsc.ar(\freq.kr(440)) }, connectionTime:3).play(chans:2)

//LFO
b = AlgaNode({ SinOsc.kr(2) })

//Connect. Notice the interpolation from 440 to the LFO over 3 seconds
a.from(b, \freq, scale:[330, 550])

//Replace b from an LFO to a random ramp generator. Notice how now a's \freq parameter is getting
//interpolated from the LFO to the newly set random ramp generator, with the same scaling
b.replace({ LFNoise1.kr(5) })

//Now replace it with a Phasor. Again, it will interpolate from the noise generator to the
//new Phasor over 3 seconds, which is a's connectionTime. Same scaling is applied
b.replace({ LFSaw.kr(5) })

//Let's change scaling
a.from(b, \freq, scale:[100, 1000])

//And replace back with a faster noise than before
b.replace({ LFNoise1.kr(10) })
```

Figure 53: A slightly more complicated example of the interpolation behaviour.

What are the technical issues that were required to be addressed in order to implement the `AlgaNode` class? First, an `AlgaNode` contains a number of elements that have to perform in sync with each other so that the system can function correctly. These elements are split between *language-side* metadata about the state of the node, and *server-side* information about running `Groups`, `Synths` and `Busses`. Both of these different kinds of data are stored in *per-parameter* `IdentityDictionaries`. An `IdentityDictionary` is a collection of *object-value* pairs in which the *object* act as a key, similarly to how *hash tables* work (Maurer & Lewis, 1975). In the case of *Alga*, the *key* is often the `Symbol` that represents a *parameter name*. The *metadata* `IdentityDictionaries` store information such as which `AlgaNodes` are connected to a specific parameter of this node (`inNodes`), and which `AlgaNodes` this node's output is connected to (`outNodes`). Furthermore, these dictionaries hold the values of per-parameter *connectionTimes* (`paramsConnectionTime`), channels mapping (`paramsChannelMapping`) and scaling values (`paramsScaling`). These all need to be stored for the `replace` mechanism to work correctly, re-using already set values when necessary.

Before talking about the `IdentityDictionaries` that store information on the state of a node on the audio server, let us first consider what is the nature of the data they hold. In order to perform audio operations on a *SuperCollider* server, `Groups`, `Synths` and `Busses` must be used. A `Group` is a 'collection of other nodes organized as a linked list' (SuperCollider, 2021). In simple terms, it is a container that can store other nodes, in this case `Synths`, that are looped through sequentially every tick of the audio scheduler. A `Synth`, then, 'represents a single sound producing unit' (SuperCollider, 2021), and a `Bus` simply describes the reference to an *audio buffer* that is available for users to be filled in with the output of their `Synths`. In fact, the way audio is streamed out of *SuperCollider* is through the first *N audio* `Busses`, where *N* is the number of outputs the server was booted with.

Alga leverages the combination of these three basic elements to describe different operations that are needed to perform the process of interpolation. An `AlgaNode` has three groups on the server: a `synthGroup`, a `normGroup` and an `interpGroup`. These groups store a different set of `Synths` which, on the language side, are handled by the `normSynths`, `normBusses`, `interpSynths` and `interpBusses` `IdentityDictionaries`.

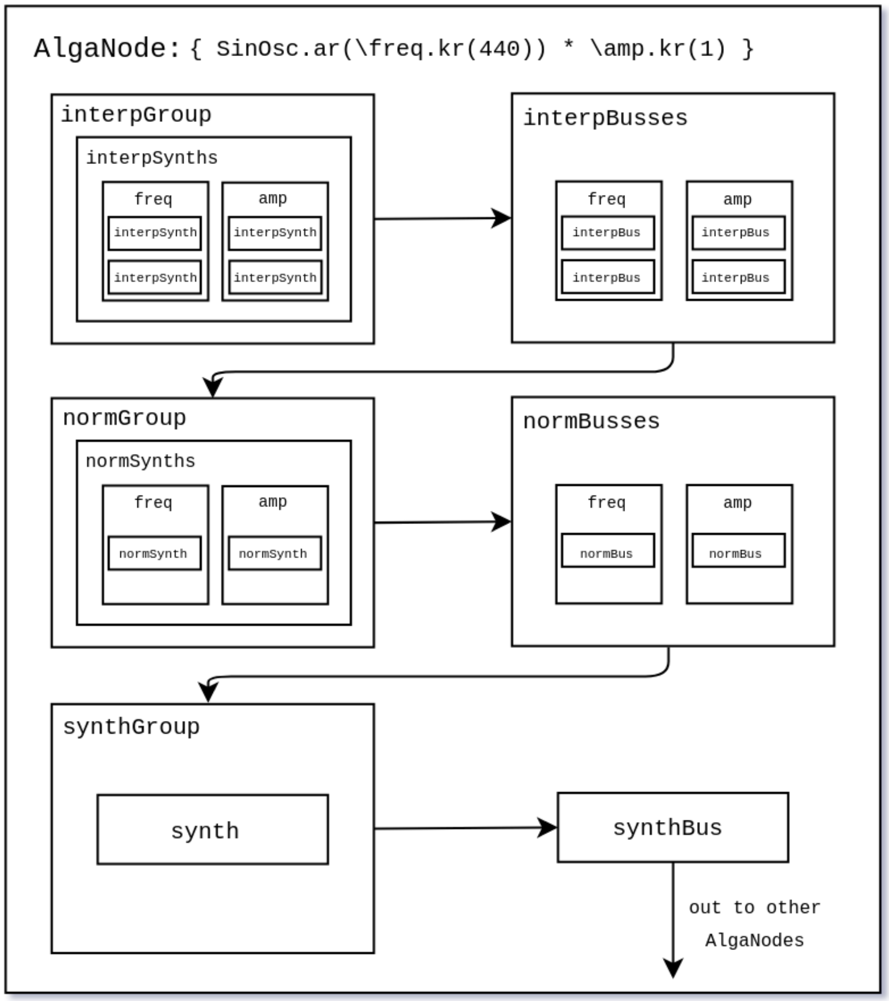


Figure 54: Anatomy of an *AlgaNode* on the *SuperCollider* server.

The `synthGroup` is used by the `synth` variable, which is the actual `Synth` that performs sound as described in the body of the `AlgaNode`. It writes its values to a `Bus` called `synthBus`. This `synth` reads the value of each of its parameters from a `Bus`, called `normBus`, that is being written by a `normSynth`. There are one `normSynth` and one `normBus` per parameter. A `normSynth` is a `Synth` instance on the `normGroup`. It reads its values from multiple `interpBusses` and *normalizes* their interpolation envelopes so that they always sum up to 1, which is fundamental for the *retriggering at any stage of the interpolation* behaviour that *Alga* proposes. Before explaining why this is the case, the `interpSynths`, which are the core of the entire interpolation behaviour, need to be introduced. As of now, there are two types of interpolation envelopes, both handled by `interpSynths`. One is a *rising* one (0 to 1), and it is triggered any time a connection with a new node is made. The other is a *descending* one (1 to 0), which is also triggered on a new connection, but it is applied to the *old* connected node. Anytime a new connection to a parameter is made, multiple `interpSynths` are active per-parameter, each one following their own envelope trajectories. An individual `interpSynth`, created on the `interpGroup`, reads from the `synthBus` of a connected `AlgaNode`, applies rate and channel conversions, multiplies the result with a *rising / descending* envelope and writes its output to an `interpBus`. The `interpBus`, then, consists of two elements: the converted and scaled signal, and the *rising / descending* envelope itself.

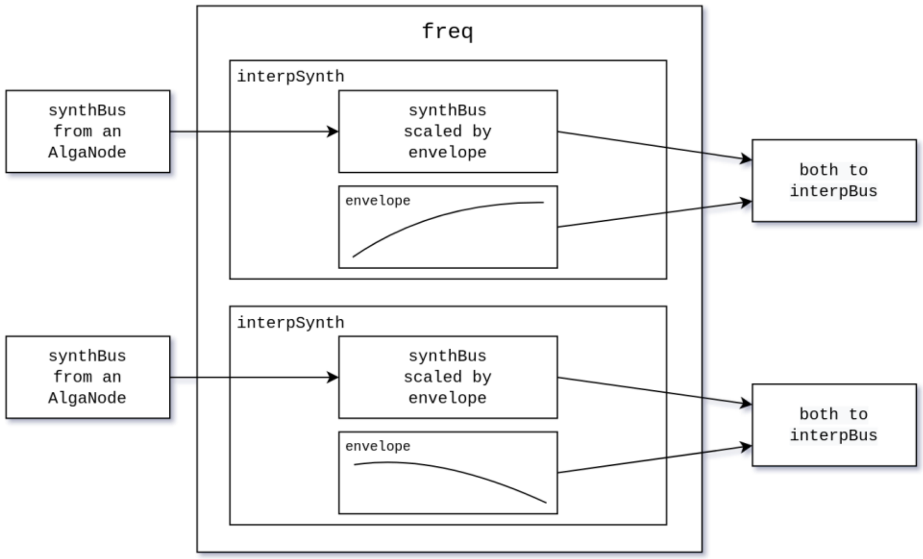

freq
interpSynth
synthBus from an AlgaNode
synthBus scaled by envelope
envelope
both to interpBus
interpSynth
synthBus from an AlgaNode
synthBus scaled by envelope
envelope
both to interpBus

Figure 55: Behaviour of an *interpSynth.*

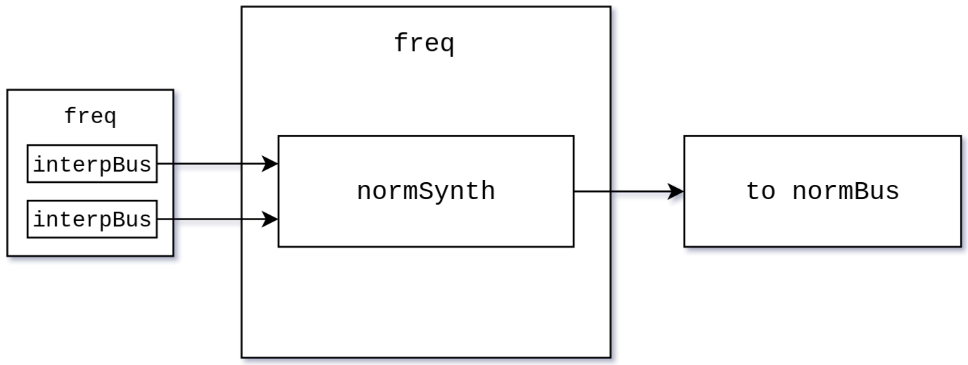


Figure 56: Behaviour of a *normSynth.*

The process of normalizing the envelope shapes in the `normSynth` is essential to never allow any envelope from any `interpSynth` to be prominent over the others. The idea is that the *rising envelopes* will dictate the direction of the interpolation, while the descending ones will only *scale down* the weight of specific nodes over the period of `connectionTime`. For the purpose of dynamic re-triggering, both envelope shapes provide a `\fadeTime` parameter that sets the speed of the interpolation at runtime. For example, if a 20 seconds long interpolation is taking a place, and after 5 seconds the user prompts a connection to a new node with a `connectionTime` of 3 seconds, the *descending* envelope that is applied to the old interpolation will be 3 seconds long, and not 20. The normalization of the envelopes in the `normSynth`, on the other hand, will make sure that all values are correct in relation to each other: the sum of the envelopes will always be 1.

In order to make sure that these actions are in sync between the language and the server, *Alga* utilizes a custom scheduling system called `AlgaScheduler`, which is the topic of the next section.

4.4.2 - *AlgaScheduler*: keeping things in order

One of the fundamental assumptions when connecting two `AlgaNodes` is that they need to be correctly instantiated on the server. In *SuperCollider* terms, the meaning of this concept is that the `Groups` and `Synths` that represent a specific `AlgaNode` must have been made *ready* to perform their calculations before allowing *nodes* to be connected to one another. Such a task should not be handled at the user level, but it should be taken care of behind the hood by the system, allowing all actions to be synchronized between *sclang* and *scsynth*. The `AlgaScheduler` class serves this purpose exactly. While it operates in the background and the user does not have to know about it, the `AlgaScheduler` is the core element that permits any *Alga action* to take place in a reliable and reproducible manner.

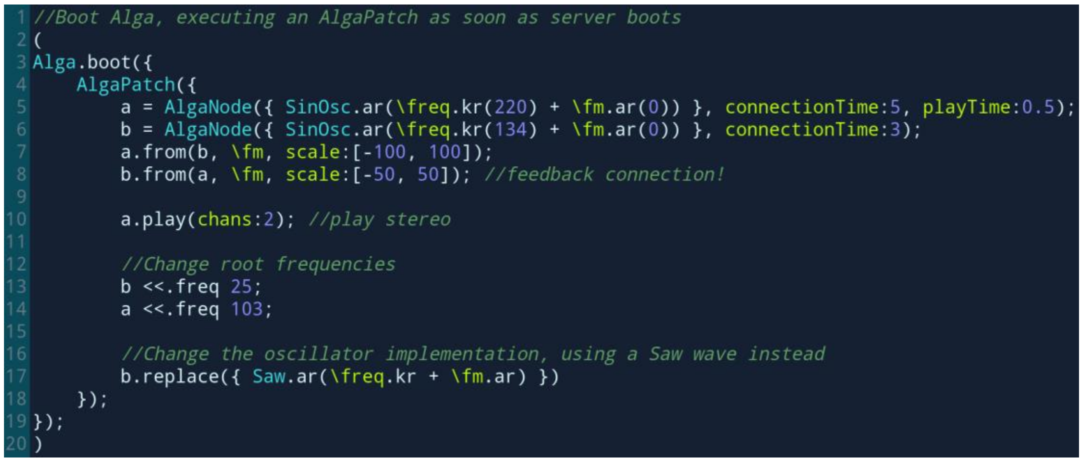
The added complexity of a system where every module can be connected to each other at any time is an issue that other *live coding* languages do not usually have to deal with. In these environments, instruments are often *isolated*, with their own chain of effects and modulations that is not affected by any other *external* stimulus. On the other hand, *Alga* needs to make sure that all access to any *node* is synchronized not only between language and server, but also between each other. In order to solve this, the `AlgaScheduler` implements a waiting queue of *actions* that is clock synced (defaulted to `TempoClock`) which can be prompted to *wait* at any stage, allowing to *chain* the execution of any *Alga* code sequentially. This feature effectively permits the creation of entire *patches* where the code execution, including the necessary synchronization, is read from top to bottom.

Internally, anytime an *Alga* function is called, a new *action* will be *pushed* to the queue of the `AlgaScheduler`. This *action* is composed of three elements: a `condition`, a `function` and a `sched` value. The `condition` is a function that is checked until it returns `true`. Once it is `true`, the `function` will be executed and the *action* will be taken off the queue. If the `condition` remains `false` after a set amount of time (which defaults to 5 seconds), the *action* will be removed from the queue and an error will be prompted on the console. This can happen if the user is trying to make connections between nodes that do not exist yet on the server. In the case of complex patches, where multiple `replace` calls happen at the same time, this behaviour makes sure that both the *language* and the *server* are synchronized when trying to express new connections. Finally, the `sched` value allows the check for the `condition`, and thus the execution of the `function`, to be *scheduled* on a `Clock`. This permits the system to synchronize multiple actions together, ensuring that they will all happen at the exact same time. In *server* terms, this signifies that the specific commands to *create or connect nodes* will have the same *OSC timestamp*, resulting in their execution to be performed at the start of the same audio buffer.

```
(
s.latency = 0.5;
Alga.boot({
    AlgaSynthDef(\sine_const, { SinOsc.ar(Rand(220, 2000)) * 0.01 } ).add;
    AlgaSynthDef(\bypass, { \in.ar } ).add;
});
)

//All OSC messages will have same timestamp
(
a = AlgaNode(\bypass).play(chans:2);
100.do { |i| b = AlgaNode(\sine_const); a.mixFrom(b, time:1); };
)
```

Figure 57: Thanks to the *AlgaScheduler*, all actions are correctly timestamped, respecting connections order.

The `AlgaScheduler` provides an argument, `cascadeMode`, that specifies one of the following two behaviours. If `false` (the default), the `condition` of the *actions* in the queue are checked in parallel, executing the relative `functions` and freeing the *action queue* accordingly. If `true`, the `conditions` are checked in order. This means that the second `condition` in the queue will not be checked until the first one is `true`, etc. This allows the `AlgaScheduler` to be extremely precise in the specifying the order in which certain *actions* are required to happen, regardless of their *scheduled* time. The `AlgaPatch` class, in fact, is a simple abstraction around this notion. An `AlgaPatch` simply takes a function that represents the user's *patch*. The `AlgaScheduler`, then, will be switched to `cascadeMode`, triggering the actions in the exact order in which they are specified, effectively reproducing the code until the last line. This allows to have *reproducible* patches everytime, with no need to re-define all the connections by triggering the individual lines of code. This example proposes two oscillators which modulate each other in feedback fashion. Calling an `AlgaPatch` makes sure to sync all the actions the same way, every time:



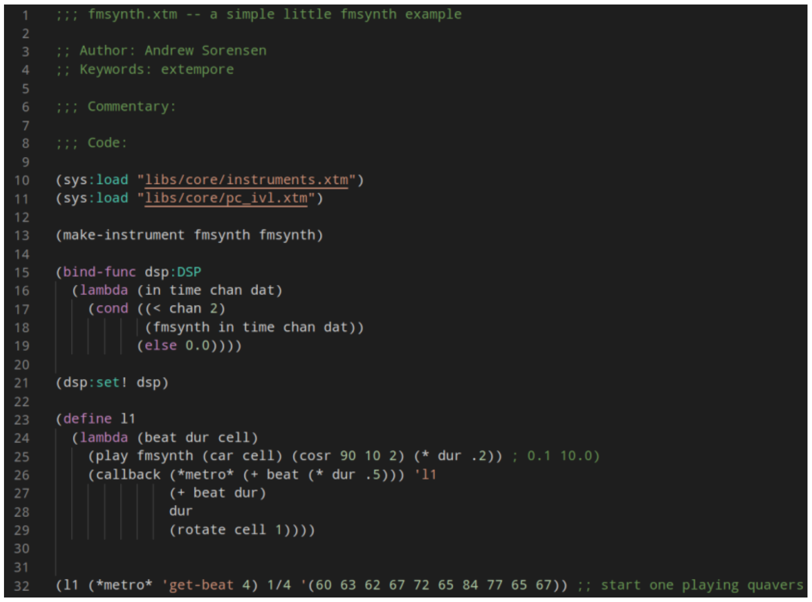
```
//Boot Alga, executing an AlgaPatch as soon as server boots
(
Alga.boot({
    AlgaPatch({
        a = AlgaNode({ SinOsc.ar(\freq.kr(220) + \fm.ar(0)) }, connectionTime:5, playTime:0.5);
        b = AlgaNode({ SinOsc.ar(\freq.kr(134) + \fm.ar(0)) }, connectionTime:3);
        a.from(b, \fm, scale:[-100, 100]);
        b.from(a, \fm, scale:[-50, 50]); //feedback connection!

        a.play(chans:2); //play stereo

        //Change root frequencies
        b <<.freq 25;
        a <<.freq 103;

        //Change the oscillator implementation, using a Saw wave instead
        b.replace({ Saw.ar(\freq.kr + \fm.ar) })
    });
});
)
```

Figure 58: An *AlgaPatch*.

Chapter 5 - Bringing the two worlds together

Both *Omni* and *Alga* are independent projects. Nevertheless, I personally use them in conjunction with each other in my musical practice. By no means the way in which I integrate them is the one correct way of using either of the two, as I believe that they are versatile enough to be individually employed in the practice of other creative coders, composers and improvisers.

Joining coding with composing is a position that is not peculiar to my practice, but it is a widespread approach among creative coders. For instance, Alex McLean and Sam Pluta have shown how developing a custom software around one's creative needs can enhance a sonic practice with new inputs, and vice-versa. The case of *TidalCycles* proved to be a successful one not only for its creator's musical practice, but also for a vast community of coders and performers (McLean, 2014). On the other hand, Pluta, as already mentioned, created a system that is custom-tailored to his own way of improvising computer music. The program he developed is the fundamental element that allows him to perform his music the exact way he envisions (Pluta, 2012). Arguably, Sam Pluta's music could not be created with any other piece of software. Such an approach to instrument development does not have to be limited by the creation of solely *high-level* software; tools that act on the structural level of sound, similarly to *Alga*. Composers like Dario Sanfilippo, in fact, developed entire compositional systems in *low-level* languages. In Sanfilippo's case, this is *FAUST*. In his case, the choice of a language like *FAUST* was dictated by his interest in developing *Complex Adaptive Systems (CASes)*, which heavily rely on the use of complex feedback interactions. A *low-level* language like *FAUST*, then, provided him with the possibility of *single sample feedback* processing, which is essential to develop the accurate filter algorithms that are required for *CASes*. Coding with *low-level* concerns, then, assumed the same value as the process of composing: the code itself *becomes* the composition (Sanfilippo, 2020). There also are projects that explicitly embrace the dicothomy between the *low-level* and *high-level* stages of the audio stack, leveraging them for both technical and creative purposes. One such case is the *Extempore live coding* programming lanugage (Sorensen, 2018), developed by the coder and improviser Andrew Sorensen. The novelty of this language resides in providing users with a unified interface across *high-level* structural concerns and *low-level* algorithm development. In *Extempore* the two ends of the spectrum share the same *Scheme-like* (Dybvig, 2009) syntax within a shared runtime environment, with minor differences in regard to memory handling and types annotation for the *low-level* part. *Extempore*, then, is a *live coding* tool that can appeal different kinds of improvisers and performers, as it allows them, depending on their expertise, to design their *live instruments* at both levels of abstraction.



```
;;; fmsynth.xtm -- a simple little fmsynth example

;; Author: Andrew Sorensen
;; Keywords: extempore

;;; Commentary:

;;; Code:

(sys:load "libs/core/instruments.xtm")
(sys:load "libs/core/pc_ivl.xtm")

(make-instrument fmsynth fmsynth)

(bind-func dsp:DSP
  (lambda (in time chan dat)
    (cond ((< chan 2)
           (fmsynth in time chan dat))
          (else 0.0))))

(dsp:set! dsp)

(define l1
  (lambda (beat dur cell)
    (play fmsynth (car cell) (cosr 90 10 2) (* dur .2)) ; 0.1 10.0)
    (callback (*metro* (+ beat (* dur .5))) 'l1
              (+ beat dur)
              dur
              (rotate cell 1))))


(l1 (*metro* 'get-beat 4) 1/4 '(60 63 62 67 72 65 84 77 65 67)) ;; start one playing quavers
```

Figure 59: *Extempore* code example

In designing my own tools, I prefer to separate the different concerns that the two levels present with syntactic solutions that leverage the strengths of the context in which they are used. In my view, the difference in time representation between the *low-level* and *high-level* stages of audio processing resemble the ones that Curtis Roads analyzes in his seminal work *Microsound*. Here Roads proposes to approach music as a product of different *time scales*, ranging from the smallest *micro-structure* of individual *sonic objects* up to the largest *macro-structure* of the *form* of a composition (Roads, pp. 3-4). *Omni* and *Alga* each represent one of the two stages. Thus, they propose solutions that I believe are functional to the *time scale* they individually constitute in the audio stack. While there are clear syntactic technical differences between the two projects, in my creative work I see a clear continuum between the high-level structuring of audio algorithms with *Alga* and the low-level development of such algorithms via *Omni*. In my case, the two poles share a common middle-ground: the *SuperCollider* language. My workflow involves creating audio objects in *Omni*, compiling them with the *OmniCollider* wrapper, and then using the resulting *UGens* in my practice with *Alga*. This hybrid approach to improvisation and composition is a crucial element of my endeavours as a creative coder, as it stimulates my musical thinking on multiple levels, from the lowest to the highest one. Moreover, this allows every aspect of my work to influence one another: composing does not stop when coding, just as coding does not end when composing. The way I tackle such concerns is to treat the *low-level* implementations of specific algorithms with *Omni* as *nodes*, *grains* of a larger *macro structure*, arranged within the *Alga* environment. Usually, the music I improvise involves *chaotic feedback oscillators* being fed into each other to create sonic movement not through explicit *modulations*, but from the *emergence* resulting from *complex interactions* between *simple entities*. In this regard, *Omni* allows me to experiment with the inner implementations of the chaotic oscillators with various algorithms, usually adapted from the nonlinear dynamics of *strange attractors* (Ruelle, 1995). On the other hand, *Alga* is helping me to *play* with these *noise structures* by *dynamically patching* them together, disclosing how their feedback interactions work not in the moment of creating a *connection* from one to another, but in the process of *interpolating* the different states over long periods of time.

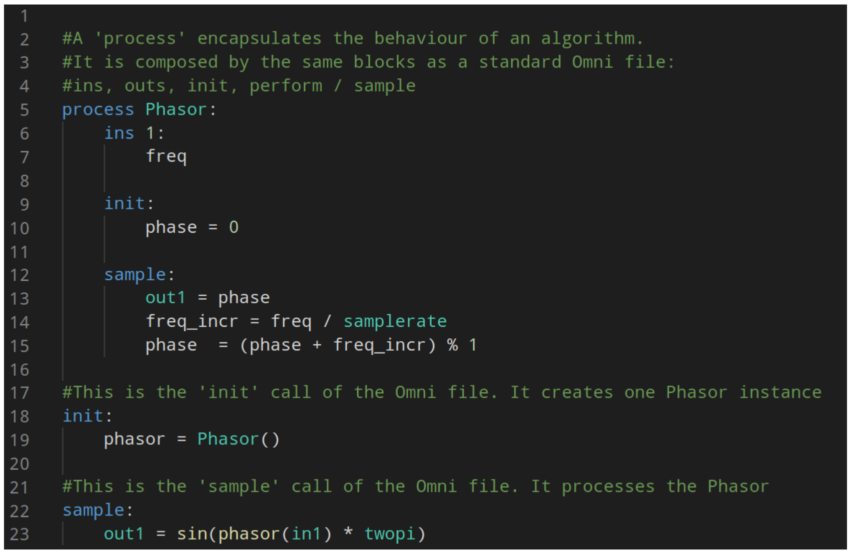
Chapter 6 - Future work and conclusions

6.1 - *Omni*'s roadmap

Omni is a project that will continue beyond the scope of my Post Graduate degree. There are many features which I am still working on, or have not been implemented yet. Currently, I am working on a *JIT* version of the *Omni* compiler, named *OmniJIT*, in order to make it compile code on the fly, instead of having users call into the compiler themselves whenever they introduce a change to their code. This would allow programmers to easily embed the entire *Omni* compilation pipeline in their own projects. The development of *OmniJIT* is also bringing some unexpected benefits that will be applied to the mainline *Omni* compiler. Mainly, these regard the removal of the need of the explicit *Nim* dependencies from the user, allowing both *Omni* and *OmniJIT* to be completely redistributable and standalone.

Additionally, two other side-projects that will be released are *OmniJUCE* and *OmniWeb*. The first one, as the name suggests, will permit users to compile *Omni* code into *VST / AU / Standalones* using the *JUCE* framework (Storer, 2004). The second one will allow *Omni* code to be compiled and run on the Web as *Web Audio Modules* (Kleimola & Larkin, 2015).

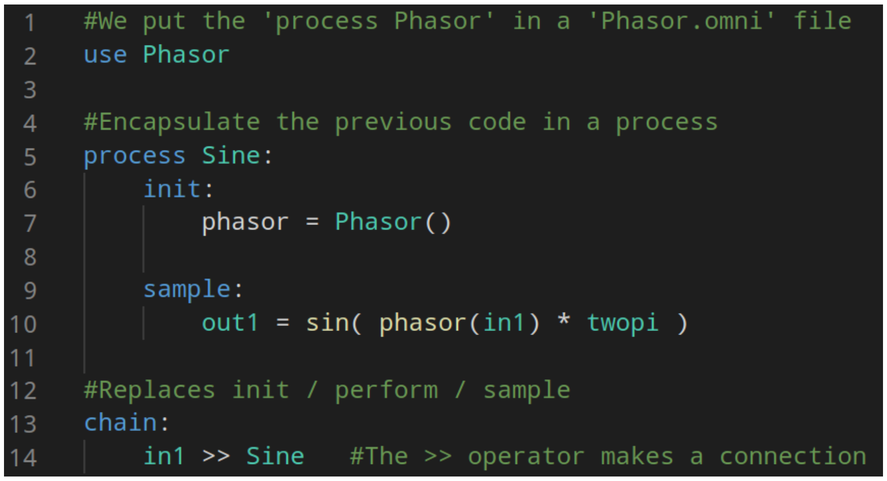
Regarding syntax, I envision *Omni* to become a *context-aware programming language* (Petricek, 2017), where each *block* not only has a unique scope, but also a unique syntax. This means that the syntax of the language itself is adaptive to the various *block contexts*. I would want to implement two new blocks that highlight this idea: the `process` block and the `chain` block. The former would be a way of encapsulating certain algorithms into *audio classes* that have the same structure of an *Omni* file. In other words, the current `struct` / `def` approach to modularity would exist as a low-level interface, while the new `process` block would be the one used to define re-usable audio algorithms. This example should clarify this notion:



```

#A 'process' encapsulates the behaviour of an algorithm.
#It is composed by the same blocks as a standard Omni file:
#ins, outs, init, perform / sample
process Phasor:
    ins 1:
        freq

    init:
        phase = 0

    sample:
        out1 = phase
        freq_incr = freq / samplerate
        phase  = (phase + freq_incr) % 1

#This is the 'init' call of the Omni file. It creates one Phasor instance
init:
    phasor = Phasor()

#This is the 'sample' call of the Omni file. It processes the Phasor
sample:
    out1 = sin(phasor(in1) * twopi)
```

Figure 60: Example of a *process* block

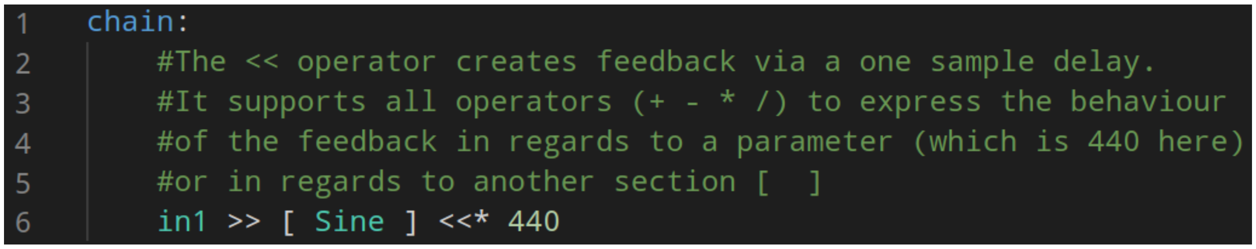
By itself, the behaviour of the *process* block can currently be implemented with `structs` and `defs`. However, paired with the `chain` block, a new mode of defining algorithms can be implemented. The `chain` block would act as an alternative to the `init` / `perform` / `sample` blocks. This new block would propose a *functional* style of patching different algorithms together, instead of the current *sample-by-sample* approach. The previous example, using the `chain` block, could be rewritten as follows:



```
#We put the 'process Phasor' in a 'Phasor.omni' file
use Phasor

#Encapsulate the previous code in a process
process Sine:
    init:
        phasor = Phasor()

    sample:
        out1 = sin( phasor(in1) * twopi )

#Replaces init / perform / sample
chain:
    in1 >> Sine   #The >> operator makes a connection
```

Figure 61: Example of a *chain* block

The `chain` block, then, could be used to express an algorithm as a series of connections between `processes` going left to right. In order to express feedback and delays, the connections can be reversed from right to left, using square brackets to delimit sections:

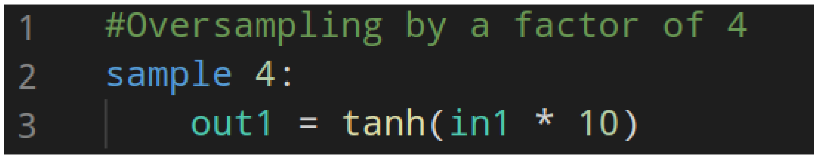


```
chain:
    #The << operator creates feedback via a one sample delay.
    #It supports all operators (+ - * /) to express the behaviour
    #of the feedback in regards to a parameter (which is 440 here)
    #or in regards to another section [  ]
    in1 >> [ Sine ] <<* 440
```

Figure 62: Proposed syntax for feedback operations in the *chain* block

The implementation is still highly experimental, and it needs a lot of testing. The reasoning to propose such a syntax is to ease the process of developing algorithms even more. One would use a `process` to specify the *sample-by-sample* behaviour of a single entity. Then, there would be the choice of defining how this `process` interacts with other `processes`. This is achieved by either using the default `init` / `perform` / `sample` syntax, dealing with *sample-by-sample* programming which can be more useful for certain algorithms like filter development, or the *patching* syntax with the `chain` block, which can perhaps be more creatively inspiring to develop things like chaotic systems and oscillators.

Additionally, together with many optimizations to the code, I would also want to implement a specific syntax for *oversampling* (Smith, 2002):



```
#Oversampling by a factor of 4
sample 4:
    out1 = tanh(in1 * 10)
```

Figure 63: Proposed syntax for oversampling in *Omni*

To conclude, another goal would be to develop support for *multi-rate* processing, especially regarding *FFTs* (Boulanger & Lazzarini, 2010, chapter 7). However, as of now, I personally do not have a clear idea in my mind on how to implement it or how to make it fit into the *Omni* syntax scheme.

6.2 - *Alga*'s roadmap

As for *Omni*, I do intend to keep developing *Alga*. The first improvement is going to be the implementation of a new class, `AlgaPattern`. This is an experimental feature that is currently being actively developed. The idea behind it consists in applying the same *interpolation* behaviour of `AlgaNodes` to the manipulation of pattern-based instruments. These require many concerns to be addressed, especially regarding the interpolation of the *triggering* of the pattern. In fact, while the interpolation of pattern parameters can be achieved with a similar implementation as the interpolation of parameters of an `AlgaNode`, interpolating the intervals at which sound is produced proved to be a more complicated problem to solve. This issue mainly arises from the fact that it is difficult to guarantee that a pattern will be in sync with the others during and after the process of interpolation. This would require an additional implementation in the `AlgaScheduler` in order to *fluidly sync* all patterns together, regarding of their interpolation state.

Another improvement that will be implemented is the possibility for the user to specify any kind of *interpolation function*. As of now, this function is a simple half-cosine function that goes either upwards or downwards, over a specific duration. It would be interesting to provide users with an interface to define custom *interpolation points* for the *upwards interpolation*, which would then be *inversed* to implement the *downward* counterpart. Perhaps, this interface could be worked around the `Env` class. In fact, the `Env` class allows users to define specific points of an envelope. It seems like the perfect candidate for this implementation.

Finally, the bigger improvement to *Alga* will be the development of a custom *live coding language* that will translate its syntax to calls into the *AlgaLib*'s *sclang* implementation. I envision this project to be developed in *Nim*, following what I learnt by developing *Omni*. I imagine the interface to be extremely minimal, with few operators to specify all of *Alga*'s features: defining and replacing a node, establishing connections, specifying parameters, etc… My idea is to develop the language as a *REPL* process (van Binsbergen et al., 2020), similarly to *TidalCycles*, that will be the only interface the user would have to deal with. This would bring two benefits: first, it would not require the coder to have to deal with the *SuperCollider* interface, which can be very verbose for specific operations. Secondly, by having a *REPL*, it would be quite trivial to embed the process in any text editor of the user liking, sending specific lines of code to the running process to be interpreted. I do not have a specification in mind yet for how the syntax should look like, as I would like to complete the *AlgaLib* implementation with `AlgaPatterns` before focusing on the actual *live coding* language.

6.3 – Conclusions

With this research project I introduced two new *DSLs*, *Omni* and *Alga*, which both serve a different purpose. The former allows users to define *low-level* algorithms and compile them to native binaries for any operating system. The latter is a *live coding* dialect within *SuperCollider* that proposes a new way of structuring sound objects by interpolating the connections between them. I have also presented the reasoning for their existence, which is my practice as coder, composer, and improviser. In fact, they both allow me to not only determine *what* music I want to create, but also *how* I want to make it. As a creative coder, this is an invaluable source of inspiration that propels any of my creative outputs.

In their current state, both projects present limitations which will be addressed in the future, as mentioned in the previous two subsections. In the case of *Omni*, I believe that the current biggest limit is the absence of a *JIT* compiler, which would be essential to further increase the rapid development process. This is a feature that is present in other *DSLs* like *gen~* and *FAUST*, and it would increase the appeal towards *Omni* for creative coders that do not want to explicitly deal with *CLIs* (Command Line Interfaces) to compile their algorithms. Regarding *Alga*, there currently are two important limitations: the absence of the notion of patterns and the limitations of being embedded in the *SuperCollider* environment. Firstly, the introduction of the notion of *AlgaPaterns* will surely be beneficial to provide users with an environment that would be more complete to fit any *live coding* need. Secondly, the development of a *standalone* language on top of *AlgaLib* will allow users to interface with a *syntax* that is designed from the ground up to forward all the concepts that *Alga* proposes, without the need of using the intermediate *SuperCollider* stage and adapt to its rules.

I believe that both projects can have an impact on other people's practice as creative coders. *Omni* proposes new ways to tackle the development of *low-level audio algorithms* with a minimal and easy to learn syntax, without compromising on performance and features' depth. These can be valuable characteristics for beginners and expert *DSP* developers alike. *Alga*, by taking a different approach to *live coding*, can help other musicians to develop a practice that draws inspiration from the world of modular synthesizers, with the unique feature of the *interpolated patching*. This single mode of operation separates *Alga* from other audio software, placing it in the position of being an *unicum* in the current landscape.

Bibliography

Bezanson, J., Karpinski, S., Shah, V. B., & Edelman, A. (2012). Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*.

Boulanger, R., & Lazzarini, V. (2010). *The Audio Programming Book*. MIT Press.

Brown, A. R. (2016). Performing with the other: the relationship of musician and machine in live coding. *International Journal of Performance Arts and Digital Media, 12(2)*, 179-186.

Burland, K., & McLean, A. (2016). Understanding live coding events. *International Journal of Performance Arts and Digital Media, 12(2)*, 139-151.

Cameli, F. (2019). JuliaCollider - Julia's JIT compilation for low-level audio synthesis and prototyping in SuperCollider. [Computer software]. https://github.com/vitreo12/JuliaCollider

Cocker, E. (2016). Performing thinking in action: the meletē of live coding. *International Journal of Performance Arts and Digital Media, 12(2)*, 102-116.

Collins, N., McLean, A., Rohrhuber, J., & Ward, A. (2003). Live coding in laptop performance. *Organised sound, 8(3)*, 321-330.

Cycling ’74. (2011). gen~ [Computer software]. https://docs.cycling74.com/max7/vignettes/gen_overview

Dybvig, R. K. (2009). *The Scheme programming language*. MIT Press.

Eldridge, A., & Kiefer, C. (2017). Self-resonating feedback cello: interfacing gestural and generative processes in improvised performance. In *NIME* (pp. 25-29).

Filip, K. (2005). ppooll [Computer software]. https://ppooll.klingt.org/index.php?title=manual

Flanagan, D., & Matsumoto, Y. (2008). *The Ruby Programming Language: Everything You Need to Know*. " O’Reilly Media, Inc.".

Fraser G. (2020). Bacalao - Somewhat fishy live cod(e) extensions to SuperCollider [Computer software]. (0.8.31). https://github.com/totalgee/bacalao

Goldman, A. (2019). Live coding helps to distinguish between embodied and propositional improvisation. *Journal of New Music Research, 48(3)*, 281-293.

Grame CNCM. (1982). *Grame CNCM*. https://www.grame.fr/

Hayes, L. S. (2014). *Audio-haptic relationships as compositional and performance strategies* (Doctoral dissertation, University of Edinburgh).

Harker, A. (2017). FrameLib: Audio DSP using frames of arbitrary length and timing. In *43rd International Computer Music Conference and the 6th International Electronic Music Week: Hearing the Self* (pp. 271-278). Shanghai Conservatory of Music.

Kelley, A. (2016). Zig Programming Language [Computer software]. https://ziglang.org/

Kirkbride, R. (2015). FoxDot [Computer software]. https://foxdot.org/

Kleimola, J., & Larkin, O. (2015). Web audio modules. In *Proc. 12th Sound and Music Computing Conference*.

Lattner, C., & Adve, V. (2004, March). LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. (pp. 75-86). IEEE.

Lilis, Y., & Savidis, A. (2019). A survey of metaprogramming languages. *ACM Computing Surveys (CSUR), 52(6)*, 1-39.

Liskov, B., & Guttag, J. (1986). *Abstraction and specification in program development* (Vol. 180). Cambridge: MIT press.

Magnusson, T. (2011). The ixi lang: A supercollider parasite for live coding. In *ICMC*.

Magnusson, T., & MCLean, A., (2018). Performing with Patterns of Time. In *The Oxford Handbook of Algorithmic Music*.

Mathews, M. V., Moore, F. R., & Risset, J. C. (1974). Computers and future music. *Science, 183(4122)*, 263-268.

Matsakis, N. D., & Klock, F. S. (2014). The Rust language. *ACM SIGAda Ada Letters, 34(3), 103-104*.

Maurer, W. D., & Lewis, T. G. (1975). Hash table methods. *ACM Computing Surveys (CSUR), 7(1)*, 5-19.

McCartney, J. (1996). SuperCollider: a new real time synthesis language. In *Proc. International Computer Music Conference (ICMC'96)* (pp. 257-258).

McLean, A. (2014, September). Making programming languages to dance to: live coding with tidal. *In Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design* (pp. 63-70).

Noble, J. (2009). Programming interactivity: a designer's guide to Processing, Arduino, and OpenFrameworks. " O'Reilly Media, Inc.".

Orlarey, Y., Fober, D., & Letz, S. (2009). FAUST: an efficient functional approach to DSP programming. In *New Computational Paradigms for Computer Music* (pp. 65-96).

Petricek, T. (2017). *Context-aware programming languages* (Doctoral dissertation, University of Cambridge).

Pluta, S. F. (2012). *Laptop Improvisation in a Multi-Dimensional Space* (Doctoral dissertation, Columbia University).

Puckette, M., & Zicarelli, D. (1990). Max/msp. *Cycling 74*, 1990-2006.

Puckette, M. S. (1997, September). Pure data. In *ICMC*.

Puckette, M. (2014, June) - The Deadly Embrace Between Music Software and Its Users. In *Proceedings of the Electroacoustic Music Studies Network Conference. Electroacoustic Music Beyond Performance*, Berlin.

Reas, C., & Fry, B. (2006). Processing: programming for the media arts. *Ai & Society*, 20(4), 526-538.

Roads, C. (2004). *Microsound*. MIT press.

Roberts, C. (2016). Code as information and code as spectacle. *International Journal of Performance Arts and Digital Media, 12(2)*, 201-206.

Rohrhuber, J., & de Campo, A. (2011). Just in time programming. In *The SuperCollider Book*. MIT Press, Cambridge, Massachusetts, (pp. 207-236).

ROLI. (2009). *ROLI*. https://roli.com/

Ruelle, D. (1995). Strange attractors. *Turbulence, Strange Attractors, and Chaos, 16*, 195.

Rumpf, A. (2008). Nim Programming Language [Computer software]. https://nim-lang.org/

Sanfilippo, D. (2020). *Complex musical behaviours via time-variant audio feedback networks and distributed adaptation: a study of autopoietic infrastructures for real-time performance systems* (Doctoral dissertation, University of Edinburgh).

Sayer, T. (2016). Cognitive load and live coding: a comparison with improvisation using traditional instruments. *International Journal of Performance Arts and Digital Media, 12(2)*, 129-138.

Schnell, N., Röbel, A., Schwarz, D., Peeters, G., & Borghesi, R. (2009, August). MuBu and friends–assembling tools for content based real-time interactive audio processing in Max/MSP. In *ICMC*.

Smith, J. O. (2002). Digital audio resampling home page. *[Online] http://www-ccrma.stanford.edu/~jos/resample.html*

Sorensen, A. C. (2018). *Extempore: The design, implementation and application of a cyber-physical programming language.* (PhD Thesis, The National Australian University).

Srinath, K. R. (2017). Python–the fastest growing programming language. *International Research Journal of Engineering and Technology, 4(12)*, 354-357.

Steele, G. (1990). *Common LISP: the language*. Elsevier.

Stefik, A., & Siebert, S. (2013). An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE), 13(4)*, 1-40.

Storer, J. (2004). JUCE [Computer software]. https://juce.com/

Storer, J. (2016). SOUL [Computer software]. https://soul-lang.github.io/SOUL/

SuperCollider. (2021). *Group | SuperCollider 3.11.1 Help*. https://doc.sccode.org/Classes/Group.html

SuperCollider. (2021). *Synth | SuperCollider 3.11.1 Help*. https://doc.sccode.org/Classes/Synth.html

Thain, D (2019). Introduction to Compilers and Language Design. *Independently published: https://www3.nd.edu/~dthain/compilerbook/compilerbook.pdf*.

TOPLAP. (2004). *TOPLAP - The home of live coding*. https://toplap.org/

van Binsbergen, L. T., Verano Merino, M., Jeanjean, P., van der Storm, T., Combemale, B., & Barais, O. (2020, November). A principled approach to REPL interpreters. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (pp. 84-100).

Van Rossum, G. (2007, June). Python Programming Language. In *USENIX annual technical conference* (Vol. 41, p. 36).

Wang, G. (2008). *The ChucK audio programming language. "A strongly-timed and on-the-fly environ/mentality"*. Princeton University.