インフラストラクチャとしての音楽プログラミング言語 *mimium* の設計

Designing *mimium*, a programming language for music as an infrastructure

松浦 知也

2022 年 3 月

概要

本論文は音楽のためのプログラミング言語 mimium の実装を通じて、インフラストラクチャとしての音楽プログラミング言語設計の理論的枠組みを構築するものである。

今日コンピューターは音楽を制作、受容するにあたってそれを通過することは必然と言えるほど身の回りに溢れているにも関わらず、その上で音楽を作るための手段としてもっとも自由度が高くコンピューターを操作できるプログラミングという手段は未だ音楽制作環境の中では主たる手段ではない。第 1 章ではそうした矛盾めいた状況を変えるための研究として、音楽のためのプログラミング言語を自らの手ですることを通して

第 2 章ではコンピューターをメタメディア装置として使う歴史と、その中でドナルド・ノーマンに代表されるコンピューターを不可視（Invisible）にしていく思想において、インフラストラクチャの重要性が対になって語られていることを起点に、科学技術社会論におけるインフラストラクチャ研究

最後にその中間点としての音楽に関わるテクノロジーの問題点と、その対抗方法として捉えられてきたサーキットベンディングやグリッチと言ったアマチュアリズムを伴うハッカー的態度には限界が近づいていることを、Oval（マーカス・ポップ）の議論を中心にして検討する。

第 3 章では音楽プログラミング言語の歴史を既存の文献を基に、前章で概観した音楽インフラストラクチャと計算機科学という大きな二つの歴史に挟まれた存在として改めて位置付ける。本稿では歴史を 50 年代、70 年代、90 年代以降に大きく 3 分割し、それぞれがラボの中のコンピューター音楽研究、チップチューンの時代、汎用プログラミング言語の理論との合流というトピックに対応する。この歴史の中で今日の音楽プログラミング言語とは数多存在するインターフェースの中から敢えて言語という形式とその特性を活かした音楽ソフトウェアと位置付けられることを示す。

第 4 章では音楽プログラミング言語の現代における特性や概念を明確にする。そのために、音楽プログラミング言語の実装の違いを Spinellis らによるドメイン固有言語のデザインパターンを参照しながら比較し、ソフトウェアとしての構成を一般化する。また、Coblenz らによる汎用プログラミング言語における評価の語彙に関する研究を参照しつつ、音楽プログラミング言語を使用するプロセスを Human-in-the-Loop モデルとして提示し、ランタイムのブラックボックスを大きくすれば動的変更に強くなる、小さくすれば表現自体の汎用性が高まる、動的変更と汎用性を両立しようとすると設計が複雑化し実装のコストが嵩むといったトレードオフが存在することを提示する。

第 5 章では筆者が設計した音楽のためのプログラミング言語 mimium の詳細を記述する。まず第 2〜4 章で示してきた背景を総括し、1. ブラックボックスを可能な限り減らす,2. 新しい表現を生むことを目的としない, 3. インフラストラクチャとしてのプログラミング言語の必要性という 3 つの設計方針を示す。mimium は型推論を伴う静的型付け言語であり、汎用の関数型プログラミング言語の設計や実装に、音楽のための言語仕様を最小限 2 種類追加する形で実装されている。1 つは関数の実行のスケジューリングを行う @ 演算子、もう 1 つはフィルターのような内部状態を伴う信号処理を通常の関数と同じよう記述可能な、状態付き関数の記法である。この 2 種類の記法を持つことで、mimium 言語上でこれまではブラックボックスとして与えられていた基本的処理の単位を

ライブラリとして (実行性能を損なわないまま) 実装可能であることを、既存の言語との比較を交えて示す。また mimium の実装に伴って新たに生じた 2 つの問題についての分析を行う。1 つは、命令型パラダイムと関数型パラダイムを同居させるために変数の生存期間の静的解析が必要になること、もう 1 つはパラメトリックにシグナルプロセッサ (関数) を複製するために、マクロや多段階計算のような、音楽プログラミングの中でこれまで非自明に扱われてきた複数の計算ステップについて考慮する必要があることである。

第 6 章では、mimium の設計自体の変遷をオートエスノグラフィーの形で記述することで、音楽プログラミング言語の実装過程

第 7 章では、前半で示した音楽プログラミング言語の存在論、歴史と後半で示した mimium の実装とその実装過程の分析をより一般化し、音楽とテクノロジーの狭間で行われる非常に個人的な動機からの問題設定を基に始めたもの（道具/メディウム/インフラストラクチャ）づくりを、それを多くの人間が共有して使うものへと開いていく研究領域を「音楽土木工学」として、近年の音楽情報処理研究との対比や計算機科学の動向、handredrabbits や ucnv といったアーティストの実践を交えて検討する。今後の研究課題として、音楽という応用領域の中では背景化されてきたオペレーティングシステムやコンピューターアーキテクチャといった計算機の根幹に関わる理論自体の見直しを行うことで、mimium の実装に伴って明らかになった問題を解決する糸口になることを示す。

謝辞

目次

第 1 章

序論

1.1　イントロのイントロ

2021 年 9 月。北九州市の植物園でこのイントロダクションを書いている。

植物園、というか自然公園みたいなこの場所にはフリーランスの仕事で来ている。公園を大きく使ったナイトウォークのようなイルミネーションのようなイベントで、そのための音楽や SE を流すためのシステムを作る仕事である。仕事の半分くらいはスピーカーをどこに配置するかとか、どの機材を使うかとかそういう話なのだが、もう半分はその沢山あるスピーカーにどう音を割り振ったり、パソコンを起動したら自動的に音が鳴り始めるようにするためのプログラムを作る作業になる。

制御のためのソフトウェアは Cycling'74 社の Max という音声処理が得意なマルチメディアプログラミング環境で構築している。長い歴史を持ち、現在音楽のためのプログラミング環境としては（知る限り）最も多く使われており、個人的には 2015 年から使い始めたこの Max というソフトウェアを使うと、例えば展示空間にスピーカーを 10 個とか（それも、いわゆるサラウンドとかとは全然違うレイアウトで）配置してそれぞれに違う音を同期して流したりすることができる。あるいは、映像を生成するコンピュータからネットワーク経由で特定の信号を受け取ったら特定の効果音を流すようにしたり。

なんだか言葉にすると大したことがない仕事のように聞こえる。実際、プログラムを書く作業自体は主観的には大したことがないのだ（ソフトウェア特有のニッチな問題やらバグは細々あるにしても）。こんな作業でお金をもらってしまっていいんだろうかと思う時もある。

しかし実際のところ似たような仕事ができる人はそこまで多いわけでもないらしい。例えば、仕事を受けようとしたが予定が埋まっているときに別の似たような職能を持った人を紹介しようと思っても、パッと思いつく人はそう多くない。

これは、自分がこうした仕事に慣れすぎてしまったのだろうか。半分はそうなのだろうが、この仕事をしているとやはり考えざるを得ない疑問は、コンピューターはなんでもできる装置のはずなのに、たかだか沢山のスピーカーに同時にたくさん音楽を流すぐらいのことに、どうしていちいちこんな専用の（しかも有料の）ソフトウェアを使ってカスタムツールを使ってプログラムを作るような手間がかかる作業になってしまうのだろう、ということなのだ。

これは別に、使うスピーカーが多いから必要なコンピューター処理能力が大きい、という話でもない。例えば、映像インスタレーション作品でどうしてもフォーマット的に 3ch のスピーカーを制御したい、となったときにも、大体 10ch の時と同じくらいの面倒臭さが発生する。2ch が 3ch になったりするだけで、普通の音楽制作ソフトウェアでは扱いづらくなる。なんで macOS でファイルを選択してスペースバーを押したら音声ファイルが再生されるように、物理的なセンサーに触ったら音声ファイルを再生する程度のことに労力が必要なのか。なんでエクセルファイルをちょっと編集するぐらいの手間で 10ch のスピーカーに音声ファイルを割り振れるようにならないのか。

この疑問こそが、おそらくはインフラストラクチャやフォーマットの持つ力というものの説明なのだろう。

つまり、人々は 2ch のステレオ音声のフォーマットがすでにあるから 2ch ステレオで音楽を作る。作るためのツールも 2ch が一番主流なのでそれが最も作りやすいように良心的な配慮をしてしまう。それに、いかに音楽や芸術表現が新しさを欲するものだとしても、とりあえずはその 2ch のフォーマットの中で表現が可能な新しさであれば問題にはならない。そして、別に 3ch 以上の音声フォーマットも表現として” 不可能なわけではない”。ただ、ちょっと手間がかかるのだ。

技術決定論者/あるいはマルクス主義者風の言い方をすればインフラストラクチャ: 下部構造が上部構造ー、つまり音楽表現とかを規定するということになるのだろうか。おそらくはそうではない。インフラストラクチャは表現を完全に縛りはしないが**誘導**する。そしてその誘導方向は既存の上部構造によって作り出されるものであり、互いに循環し合いながら徐々に路面は踏み固められていく。

1.2　イントロダクション

本論文は、音楽のためのプログラミング言語 mimium の設計と実装を通じて、音楽プログラミング言語を作るという行為の概念化を試み、その上であるべきコンピューターと音楽の関係性を考え直すものである。

本論文で議論する「音楽プログラミング言語」は、ややインフォーマルに説明すれば、テキストやグラフィック操作を用いてコンピューターで音楽を生成するための人工言語である。

その多くはコンピューターを用いて、既存の楽器では不可能な新しい音楽表現を追求するために使われたり、近年ではコードを書くプロセスそのものを演奏として見せるライブコーディングのような表現なども登場しており、プログラミングを用いた音楽表現はますます広がりを見せている。

プログラミングに限らず音楽とコンピューターの関係性に視野を広げてみれば、2021 年現在、音楽を聴いたり、演奏したり、作ったりする上で、コンピュータが一切関与しない、という状況を考えるのは難しくなっている。作曲には Protools や Cubase に代表される DAW（Digital Audio Workstation）ソフトウェアを使用し、配信には Apple Music や Spotify のようなストリーミングサービスを通じて、デジタルデータという形で音楽は配布される。最終的に、コンピューターやスマートフォン上のソフトウェアでそのデータをデコードし、DAC(Digital-Analog Converter) によって電気信号へと変換され、その信号はスピーカーへと送られようやく空気の振動になり、私たちの耳へ届く。スピーカーの中にさえデジタル信号処理 (DSP:Digital Signal Processing) 用のチップが入っていて計算によって音質の調整をしていることも珍しくはない。2020 年以後のコロナウィルスの影響も含めれば、クラシック音楽のコンサートさえもその場で空気の振動を体感することよりも録画録音されたものをコンピューターを通じて摂取することの方が多くなってしまったかもしれない。とかく音楽文化を見れば、マーク・ワイザーの提唱したユビキタス（Ubiquitous: 偏在する）・コンピューティングの概念は字義通りには達成されたようにも見える。

それにもにもかかわらず、音楽文化全体を見てみれば、音楽制作自体にコンピューターの可変性を十分に活かせるはずのプログラミングという手段を用いること自体はメインストリームとは程遠いと言える。例えば、今日の DAW ソフトウェアの中で、プログラムを用いてソフトウェアそれ自体を拡張する方法は、Ableton 社の Live における、音楽プログラミング環境 Max を内部拡張として用いることができる Max for Live、独自のスクリプト言語を使用できる Reaper などを除けば、VST プラグインなど一部の仕組みに限られる。こうした拡張も、多くは C++ のような、音楽を専門とする人には難易度の高いプログラミング言語で記述する必要があり、作家が自ら道具の機能を拡張するには未だハードルが高い。

音楽にプログラミングを用いることですらこの現状であるので、音楽のためのプログラミング言語や環境自体を作ることの事例はますます限られている。今日ソースコード共有サービス GitHub において、“Programming Language” と検索すれば 87502 のリポジトリが出てくるのに対して、“Sound Programming Language” では 137 しか出てこないことからその探究の規模の小ささががわかるだろう。実際第 3 章で見ていくことになる 2000 年代以後に開発された音楽プログラミング言語は多く数えて 30 ちょっとに限られ、年を

追うごとにコミュニティが拡大しているとも言い難い。

そして、受容の形式という視点で音楽とテクノロジーとの関わりを見てみると、どれだけ高度なテクノロジーを用いて生成された音声も、最終的には何らかの 5〜10 分の音声ファイルとして編集され、ほとんどの場合スピーカーやヘッドホンという 2ch の音波を発する装置によって発され耳に届くという、コンピューターが発明される前の 19 世紀の録音技術黎明期の受容の形式から大きくは変化していないと言える。

本論文での、音楽プログラミング言語を設計する第一の問題意識となるのは、このような、コンピューターが本来万能とも言ってもいい可変性を持っているはずで、現状音楽に関与するあらゆるマシンの中にコンピューターが関与しているにも関わらず、音楽や音を用いた表現の形式には変化が乏しいのは何故だろうかという疑問、そして、コンピューターの可変性を十全に発揮するための道具とも言えるプログラミング言語を、音楽という目的に特化させて作ることは音楽表現の新たな可能性の追求としては一見最もストレートなアプローチにも見えるのに、何故未だにその開発事例が少ないのか、という疑問である。

1.3 方法論：デザイン実践を通した研究

この疑問に対して、本研究ではデザインの実践を通した研究アプローチをとる。簡単に言い換えると、音楽プログラミング言語を設計：デザイン、また実装することを通じて、音楽プログラミングがなぜメインストリームな手段とならないのか、音楽におけるコンピューターの利用のされ方が旧来の文化様式に固定されたままなのかを明らかにするということだ。

それゆえ、本研究は通常コンピューター科学の応用分野とも言える、音楽に特化したプログラミング言語の設計の研究としては以下のような特徴がある。

1: まず、研究のアウトプットのひとつであるプログラミング言語、mimium の設計や実装はプロトタイプではなく、実際に使われることを想定している。

なぜなら、音楽プログラミング言語がなぜ多くは作られていないのかという疑問に答えるには、その言語の機関部分だけを作るーつまり、Proof-of-Work 的なプロトタイプを作るだけでは、例えば実際にソースコード共有サイトへのアップロード、リリースの作業、さらにリリース後のメンテナンスや、ドキュメンテーションといった、ツールを実際に運用するプロセスの中にも、言語を実際に作るにあたって高いハードルが存在しているかもしれなからだ。実際に、mimium の実行プログラムのソースコードはすでに Github に公開されており、何人かが開発の一部に参加してくれてもいる。

2: しかし、研究のアウトプットとしてのプログラムで実用的に役に立つほどに完成はしていなくてもよいものとする。

これは次のような理由で正当化できよう。まず、プログラミング言語やそれを実行するプログラムには明確な完成という状態を定義しにくい。常に新しい言語仕様が追加されたり削除されたりを繰り返しながら時間をかけてアップデートされていく。同じ C++ という言語でも、C++98 という初期バージョンの言語と C++20 という最新のリリースでは、言語仕様もそれで書かれるソースコードの様式も大きく異なる。

さらに、プログラミング言語の開発やそれが実際に使われるようになるまでには一般的に 2 年を超える長い時間がかかるため、実用になって初めてその内容を研究として提示するプロセスを取ってしまっては研究サイクルとして時間が掛かりすぎるという問題がある。たとえば Ruby を開発したまつもとゆきひろによれば Ruby の開発自体は 1993 年に成され、最低限の機能が揃ったのは半年後 (**Matsumoto2014**) ながら、最初に公開されたバージョンである 0.95 は 1995 年と 2 年の時間を要している。音楽系の言語の例で言えば、mimium の参考になっている言語である Faust はプロジェクト自体が始まったのが 2002 年 (https://faust.grame.fr/about/) だが、最初にリリースされた正式なバージョン 0.9.0 はやはり 2 年後の 2004 年になってリリースされている（https://github.com/grame-cncm/faust/tree/v0-9-0)。それでも研究対象としては具体的な幾つかの言語仕様に焦点を当てて議論を行うことで学術的な知見を生み出すことは十分に可

能だと言えるはずだ。実際に mimium は最初のバージョンをリリースするのに 9 ヶ月、そもそも音楽のための言語でありながら信号処理をして実際に音がなるまでに 7 ヶ月の時間を要しており、この論文が書かれている現在 (バージョン 0.4.0) でも、外部ファイルを読み込む include 機能は単なるテキスト置換で行っており、実行性能の効率も悪いし予期しない動作を引き起こしかねないという、とりあえずの間に合わせとして実装されているように、暫定的な機能やバグが多数存在する。

3: 学術的研究として、作った言語の評価としてユーザーからのフィードバックを（定量的な実験結果であれ、インタビュー等の質的な内容であれ）必ずしも必要としない。

この理由としては、筆者が焦点を当てたいのは音楽プログラミング言語がなぜ使われないかよりも、なぜ作る人の人数が増えないのかという問題であること、さらには、プログラミングという行為は、誰かの作ったライブラリのソースコードが公開されてさえいれば、その中身をどこまでも辿っていける、知識の階層を思うがままに辿っていける道路のネットワークであるにもかかわらず、コンピューターの中で音を（アプリケーションなどを通して）扱うことと、音楽や音のためのプログラムを作ること、さらには音楽や音を**記述するためのプログラム**：音楽プログラミング言語を作るということにはそれぞれ大きな隔たりがある、という問題であるからだ。この疑問に答えるために調べなければいけない対象は、すでに多く研究対象となっているユーザー：音楽のためのプログラム、アプリケーションや音楽プログラミング言語を仕様する人たちだけではなく、むしろ研究主体である自分自身とその経験なのだ。つまり音楽プログラミング言語やそれを作るという行為がブラックボックス化されている現状を、自らの手で作ることによって開き、そこで使われる知識や語彙の体系化を改めて試みることで疑問に対するヒントを掴むような道筋を辿ることになる。

もちろんこれは、1. で述べたように実用的なツールとしても mimium を開発している以上、そのツールをよりよいものにしていくために今後ユーザーにインタビューなどの形でフィードバックを求めることは有益になるだろう、という意見を否定するものにはならない。

本研究のような音楽のためのソフトウェアを制作することを研究にするにあたって、どのように研究を正当化するかに関しては、コンピューター科学の中でも Human-Computer-Interaction(HCI) の分野や、Creativity Support Tools(CST)、End-User-Programming(EUP) といった分野で議論が行われている。このようなデザインの実践を通した研究:Research through Design(RtD) としての本研究の位置付けに関しては 6 章で改めて詳しく議論するが、本研究は、何か既存の言語に足りない機能があったり、既存の言語ではできない表現があるので新しい言語を作るのではなく、言語の実装と文献調査を通して、音楽プログラミング言語という研究領域はどういった領域か、またその領域にどんな課題があるかといったものを明確化することを主眼においている。そのため、プログラミング言語 mimium 自体も特定の問題意識や仮説に基づいた設計や実装が行われてはいるものの、実装するうちにその問題意識も徐々に変化していることには留意する必要がある。第 3 章で提示する音楽プログラミング言語の歴史観や、第 4 章で議論する音楽プログラミング言語の存在論は mimium 実装のためのモチベーションであると同時に、mimium を実装することによって得られた洞察でもあるということだ。

このような考察→設計→実装→考察... といった循環的な作業を行う以上、それを問題提起→解決という線状のプロセスに形式的にでも開くことは、試行と実験を繰り返す中で発生する考えの変遷のディテールを削いでしまうことは否定できない。それを補うためにも mimium の設計と実装が時系列的にどのように行われてきたかについてをまず簡単にまとめておくことにする。

1.3.1　mimium 制作までのバックグラウンド

本研究の主題となる音楽プログラミング言語 mimium の開発は 2018 年頃に遡る。

個人的な動機の話にはなるが、筆者のバックグラウンドはコンピューターサイエンスではなく、音楽プログラミング言語を作り始めようと思ってからプログラミング言語理論を学び始めたという順番を取っており、そのバックグラウンドが音楽プログラミング言語研究の方法論自体の見直しという本論文の大きなテーマの一つ

にも大きく影響してくるのでその経緯を簡単に記しておくことにする。

2018 年の 9 月から 11 月にかけて、筆者は School for Poetic Computation(SFPC) という、ニューヨークにあるアーティスト・ラン・スクールへ留学した。SFPC はテクノロジーと表現を批評的に、かつ実践的に学ぶ私学校のような場所で、年に 2 回、20 人ほどの学生（10 代から 50 代まで幅広い）が集まり、OpenFrameworks[*1] を用いたグラフィックプログラミングから技術批評の文献購読までを 3 ヶ月間、対話を交わしながら進めていくものだった。

筆者はそれまで、物理モデリング楽器を物理的な要素で再構築するインスタレーション作品の制作や、オーディオフィードバックを主要素とする電子音響楽器の開発とそれを用いた即興演奏などを中心に活動してきた。

これらの作品制作への大元のモチベーションは簡単に言うならば、どれだけ工夫したプログラムを作ったところで全ては 2ch のステレオ PCM サウンドというフォーマットに収束してしまうことへの窮屈さから来る、より異なる音楽表現の可能性の追求であった。つまりインスタレーション作品も、オーディオフィードバックを用いる電子楽器も、コンピューターにスピーカーを繋げるだけでは実現不可能な表現領域を探索することに意義を見いだしていた。

しかし同時に、音楽のフォーマット（あるいは、より広範な意味合いでの” 形式”）から離れたところで、インスタレーションや、展示、あるいは即興演奏のイディオムという異なる形式へと従属する対象が変わっただけのようにも感じていた。

そうした疑問を抱えながら過ごした SFPC での授業は全くこれまでと違う考え方を自分に与えてくれた場所であった。

SFPC の入居しているウエストベス・アーティスツ・コミュニティという建物は、元々 1890 年代から 1960 年代までベル研究所の建物だった場所で、有名なところではショックレーらによる切開で初めてのトランジスタが発明された場所でもある。そして同時にこの場所はニューヨークのテクノロジー・アートの歴史の中心地と言ってもよい歴史的な経緯を持つ場所でもある。

ベル研究所のエンジニア、ビリー・クルーヴァーは美術家のロバート・ラウシェンバーグとともに、60 年代に Experiments in Arts And Technology(E.A.T) というアーティストとエンジニアの共同作業のための集団を組織し、《九つの夕べ》や、1970 年の大阪万博ペプシ館の演出を担当するなど、のちのテクノロジーを用いた芸術制作へと大きな影響を与えている。こうしたニューヨークにおける技術者と芸術家の共同作業は、ラウシェンバーグを始め、ウエストベスにのちに入居するダンサーのマース・カニングハムや、彼らとの共同作業も行った作曲家のジョン・ケージやデイヴィッド・チューダーといったアーティストの参加したブラックマウンテン・カレッジ、またケージの参加したフルクサスのような、反制度、反形式的な思想や運動を背景としていることが特徴であり、これは例えば人類学者のジョージナ・ボーンが”Rationalizing Culture” で、フランスの電子音楽研究所において 70˜80 年代にピエール・ブーレーズのような作曲家が電子音楽のための技術を、現代音楽という積み重ねられてきた歴史の上で正当化する過程を描いた様子とは大きく異なる (Born 1995)。

SFPC のプログラムはこうした歴史的背景をもとに、ブラックマウンテン・カレッジのような既存の教育の枠を外し、生徒が互いに教え合う機会を多く設けた”Horizontal Pedagogy”(http://taeyoonchoi.com/2012/05/notes-on-critical-pedagogy/) と呼ぶ学びの姿勢を重視したものであり、かつ、大阪万博を一つのピークとして捉えられる、60 年代のテクノロジーアートにおける、表現に先行して技術を無批判に用いることや、大規模化するごとに資本主義に迎合していってしまう傾向[*2]への内省を踏まえたものとなっている。技術を用いる際に暗黙的に発生する政治性に自覚的になり、また技術を与えられたブラックボックスとせず、すでにライブラリやツールが存在しているものだったとしても一度自分の手で作り直すことによって体でその内容を理解すること

[*1] openframeworks

[*2] 例えば、(馬 2014, 58p) の坂根厳夫の引用では、70 年代はじめのオイルショックや環境問題を単に発した科学技術自体への批判との共鳴が指摘されているし、大阪万博ペプシ館でのスポンサーによる会期中のプログラム中断と、E.A.T のその後の活動の衰退をあげることができるだろう。

で、異なる技術のエコシステムの可能性を想像することができるようになるというわけである。

それゆえ、SFPC で教える講師達の活動形態も、必ずしも作品を作って美術館に展示することが中心なわけではない。OpenFrameworks でほぼ毎日短い CG アニメーションを作り投稿し続ける Zachaly Lieberman や、CPU の動作原理を餃子作りに見立てた”CPU Dumpling” ワークショップを行う Taeyoon Choi、ポストコロニアルスタディーズをベースに、白人中心主義的なコンピューター文化批判のインスタレーションや Zine を制作する American Artist、100 年以上ただ時を刻むだけのカウンターのようなオーバー・エンジニアリングなツールを自宅の工房で作って自分たちで売り続ける CW&T. . . . . のように、多様な活動を見せる彼/彼女らの活動を通して（それでどうやって生計を立てられるかはともかくとしても）テクノロジーと社会の関係性への向き合い方は必ずしもアートという閉じられた空間の中だけで行うものでなくてもよいのだと実感させられた。

こうした経験が音楽プログラミング言語という実に微妙な領域の制作へ筆者が本格的に入っていくきっかけとなった。

詰まるところ、はじめにあげた” どれだけ工夫したプログラムを作ったところで全ては 2ch のステレオ PCM サウンドというフォーマットに収束してしまうことへの窮屈さ” に対して真に向き合うには、何が社会的にそのフォーマットを構築しているのかについて突き詰めて考える必要があったのだ。違う表現の形式へスライドしても結局その形式の束縛を受けるだけになってしまう。ならばやるべきは制作を無意識的に支配している形式や制度そのものの脱構築に他ならない。つまり筆者にとって音楽プログラミング言語の制作とは音楽を生み出す下部構造＝インフラストラクチャになりうる道具自体を自らの手で作ることを通じて、異なる音楽の形式や制度そのものをメタ的に制作する芸術（と言い切ってしまうことにより失われるニュアンスがあり他の言葉を探している途中だが、暫定的に芸術と言ってしまう）実践でもある。

この大きな問題意識のもと、筆者の音楽プログラミング言語制作の目標はその中で二転三転を経て mimium という現在のプロジェクトに落ち着く。始めにある程度まとまった人数に対して言語設計の話を持ち出したのは、SFPC 滞在期間中の自己紹介兼、3 ヶ月の間に何をやろうとしているかのプレゼンテーションをする場所だった[*3]。そこでは、1 つのソースコードに対して複数の処理系を通すことで異なる音楽が発生するという、どちらかといえば Code Poetry[*4]などの詩としてのソースコードとその処理系の関係性、コードを書く人と処理系を作る人のオーサーシップの関係性に焦点を当てたものだった。結局 SFPC の滞在期間中に制作したのはよりハードウェアのレイヤーでコンピューターの構造と音楽という時間芸術の形式の関係性を問い直すものになり[*5]、言語設計を実際に始めるのはもう少し後になる。

2019 年ごろの設計初期段階では、Mayer、Chugh らの Sketch-n-Sketch(Mayer, Kuncak, and Chugh 2018) や、Jacobs らの Para(Jacobs et al. n.d.)、橋本麦による Glisp(NiU)[*6]など、グラフィック生成のためのアプリケーションにおいて、ソースコードの編集を Direct Manipulation(Adobe Illustlator や Microsoft PowerPoint におけるベクター図形編集のような、図形の要素を GUI で直接サイズ調整できるような種類のアプリケーション) と組み合わせたり、相互に行き来できるソフトウェアを、音楽プログラミングの領域においても実現できないかというアイデアが中心に置かれていた。

実際、この提案は 2019 年情報処理推進機構未踏 IT 人材発掘・育成事業という、革新的なソフトウェア制作に対する支援事業に採択され、その当初のアイデアはプログラミング言語設計とそのソースコードをグラフィカルに編集できるソフトウェアという 2 つのプロジェクトを並列して行うものになっていた。[*7]

[*3] SFPC 2018 fall class における Meet the Students という公開イベント。Youtube の URL と SpeakerDeck のスライドを貼る

[*4] プログラミング言語のソースコード自体を詩のように扱う芸術の形式。実際には実行できないがソースコードの見た目を模したもの、実行することで初めて詩として読めるもの、ソースコードとしても可読性があり、実行することでさらにグラフィックを生成するものなど様々な種類がある。(**Montfort2013**) などを参照。

[*5] Electronic Delay Time Automatic Calculator(EDTAC) という作品。修士論文 (松浦 2019) や 2019 年 JSSA 研究会での (松浦 and 城 2019) を参照。

[*6] https://glisp.app

[*7] 未踏 IT 人材発掘・育成事業：2019 年度採択プロジェクト概要（松浦 PJ）

結果として、2019 年 6 月から 2020 年 2 月までの同事業でのディスカッションを中心に実装を進める中で、実装内容はその根幹となる、音を生成するプログラミング言語一本に絞っていくことになった。その理由は主に、筆者がプログラミング言語実装そのものの領域に対しては初心者であったが故に実装にかかる時間的な都合の問題が主な理由であった。言い訳がましく聞こえるかもしれないがこれは同事業で言語設計、実装を進めた中での大きな収穫の一つだと考えている。つまり、**音楽プログラミング言語を設計したり実装することは、想像しているよりもずっと難しかった**という、本論文の 1 つのテーマでもある、なぜ音楽プログラミング言語を作ることのハードルが高いのかという疑問に答えるための材料を身を持って入手したわけである。

1.4 Problem Statement

本研究は以上のような動機をもとにしているため、音楽プログラミング言語の開発が主題ではあるものの、その言語を作ることによって何かユーザビリティが改善されたり、それまで不可能だった表現が可能になるといった具体的な効用を実証するものではない。そうではなくむしろ、作られた mimium という言語は 2021 年現在において、音楽プログラミング言語とはなんなのか、あるいは音楽プログラミング言語をつくるとはどういう行為なのかという疑問を観察するためのレンズのようなものと考えてもらいたい。

このような、人文学的な視点からの音楽プログラミング言語制作についての研究の先行例としては、(Magnusson 2009) と (Mclean 2011) がある。

Magunusson はコンピューターを用いた楽器がアコースティック楽器とどのように異なるかを、自身の開発した IXI というライブコーディング環境での実践を例にしつつ、現象学や認識論、哲学を経由することで議論している。それによると、デジタル楽器 (DMI) はそれを構成するシステム自体がコンピューターという象徴（シンボリック）なシステムに依存しているため、より道具そのものも既存の音楽や信号処理の言語体系を反映したものになる、**認識論的道具（Epistemic Tools）**であると指摘している。

また Mclean は 〜〜説明が難しい

これら 2 つの先行研究との本研究のアプローチの違いとしては、先行研究では、アーティストがパフォーマンスややライブ演奏をするにあたり、言語や記号というシステムを介してコンピューターとインタラクションを行う過程を描き出したものだったのに対し、本研究が焦点をあてるのは、コンピューター言語

音楽言語というブラックボックスがあること（知識的な）それを開けても別の開けられないブラックボックス（UGEN）が残ること

1.5 本論文の貢献

本論文を通じて学術的に貢献する内容をまとめると、以下のようになる。

1 つ目は、音楽プログラミング言語の存在論の提示だ。音楽プログラミング言語は歴史的にコンピューターを用いて音楽を生成するための技術としてスタートしつつも汎用プログラミング言語の理論を取り込んで発展してきたことにより、一重に音楽のためのプログラミング言語/環境といってもその応用範囲や想定される使用方法、さらに内部の実装方法までが多岐に及び、単純な比較をすることが難しい。また、評価のための語彙も” 表現力が高い” “効率的” “汎用的” などの言葉が共通認識の無いまま慣例的に使われており、実際に何を意味するかもはっきりしないことがある。本稿ではまずこのような、“音楽プログラミング言語とは何か”、“音楽プログラミング言語にはどんな種類があるのか”、“音楽プログラミング言語の特性はどう記述できるか” といった概念を整理して提示する。

2 つ目は、音楽プログラミング言語を音楽を作るためのツールとしてだけではなく、インフラストラクチャ

(https://www.ipa.go.jp/jinzai/mitou/2019/gaiyou_tk-1.html) を参照。また同時期に情報処理学会音声情報処理研究会にて同様の案についてのポスター発表を行っている (松浦知也, 城一裕, et al. 2019)。

としての役割から批評することだ。

() 繋ぎ録音技術（音響再生産技術）に端を欲した音楽のフォーマットは現在空間音響技術のためのフォーマットによって、原音再現という従来の明確な一つの目標を失いつつある。その環境でコンピュータ上の表現の自由度を最大限担保するためにはプログラミング言語そのものを音楽のためのフォーマットすることが必要になってくる。

3 つ目は、音楽プログラミング言語自体を学術的な研究とする際の異なる方法論の提示だ。音楽プログラミング言語の研究自体は、人文学的な動機で研究をされつつも、基本的には作ることで問題を解決する（≒できなかった表現をできるようにする）という、形式を取ることが多かった。本論文ではそうではなく、問題がなんなのかははっきりとはわからないままに作り始めることで、事後的に問題が浮かび上がってくる、質的研究法の中で言われる Reflexivity（自己反映性[*8]）、つまり自分の研究内容によって自分自身が変化することを重視した研究である。今回の研究では、初めは音楽プログラミング言語という範囲の中での問題点という狭い視野で作り始めたものが、最終的には音楽におけるコンピューターの使い方全般の問題や、音楽流通におけるデジタルフォーマットの問題、テクノロジーにおけるブラックボックス性の是非といったより広い問題意識へと広がっていく過程をオートエスノグラフィーの形で（？）示すことで、

最後に、筆者が設計/実装した自己拡張性の高い音楽プログラミング言語”mimium” についての設計思想とその具体的な実装を提示した上で、以上 3 つの観点からの位置付けを試みることだ。mimium は先述した音楽のためのプログラミング言語の歴史を見直し、汎用プログラミング言語の設計の上に最低限の音楽のために特化した言語機能を備える構造を取ることで音楽のための言語におけるブラックボックスを減らしながらもその実装の単純さと自己拡張性の高さを同時に実現できるように設計されている。

1.6　構成

本研究はプログラミング言語制作という実践行為を通じて音楽プログラミング言語とはそもそも何か、音楽プログラミング言語を作るとはいかなる行為なのか、それがどう役に立つのか、と言った疑問を検証するものである。

5、6 章で詳しく記述することになるが、mimium の実装そのものには音楽に関わる話題がほとんど存在しない。そこにあるのはコンピューターの中で時間をどう取り扱うかといった問題や、ソースコードというテキストデータをどのようなプロセスで機械語に近い構造まで変換するかというデータ構造の変換の問題が中心となる。これは可能な限り既存の音楽の様式に依存しない形で音楽プログラミング言語の言語仕様を定義するという、2000 年以降の音楽プログラミング言語設計の一つのテーマにのっとった帰結である。なので、音楽表現の汎用性（Universality）を意識すればするほどに実装の内容は汎用プログラミング言語の理論へと接近していき、「音楽に特化した言語」から離れていく矛盾が浮き彫りになる。

この矛盾は実装をすることで初めて

つまり、第 2〜4 章で記述することになる歴史や音楽プログラミング言語の特性は mimium 実装の背景知識であると同時に、mimium を実装することによって初めて浮かび上がった痕跡でもある。

[*8] Reflexivity という言葉は再帰性や反射性という訳語があてられることも多い。本論文中では計算機科学の用語としての再帰（Recursion）との不要な混同を避けるため、また、プログラミング言語において実行環境（コンパイラやインタプリタ）の挙動自体を実行コードからメタ的に操作できる機能のことを Reflection と呼び、日本語としては自己反映言語や自己反映計算と呼ばれることもあり、意味合いとしてはより類似しているように思われるため、自己反映性という言葉を用いる。

1.6.1 三重の歴史の提示

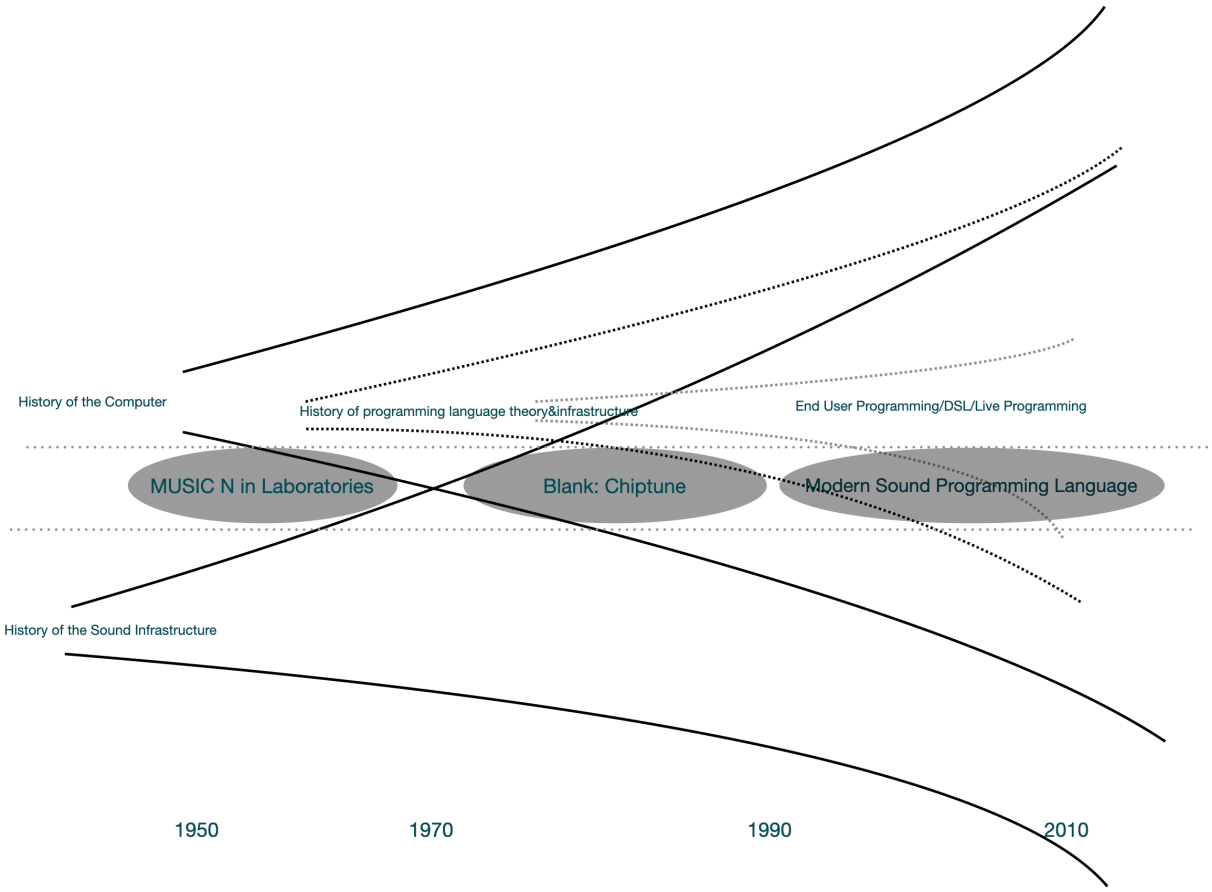


図 1.1: 本書で提示する音楽プログラミングの歴史の概要。

第 2 章以降は以下のような構成で論じる。本論文全体の構造を図 n に示した。

第 2 章ではコンピューターが単に計算の自動化のための装置から、70 年代にメタメディア装置にへと変化しさらにその先にユビキタス・コンピューティングに代表される、コンピューターを不可視（Invisible）にする思想が醸成される過程を示す。ここではメタメディアとしてのコンピューターが、ユーザーがプログラミングによって自分で機能を組み替えられるようになっていたことに本質的な点があったことを指摘する。その一方で、当時のメタメディアの音楽的な利用が既存の音楽表現のレールに乗っていることで成立していることにも言及する。

第 3 章では、現在の音楽インフラストラクチャに大きく影響を与えた録音技術

第 2 章

Why: 音楽プログラミング言語をなぜ研究するか

2.1 ブラックボックスを開く方法

> コンピューターの音楽的価値は本物の楽器にできる事を全て複製できるという能力はもちろんのこと、実際の楽器音を包括し、超えて拡張された音の種類を生み出せることにあります。(Mathews & Risset, 1969, 筆者訳)

> バイオリン、ギター、フルート、ピアノが合わさった楽器を想像してみよう。そんなことができるだろうか。もちろん。小粋なシンセサイザープログラムがパソコンキーボードからこれら全ての楽器の音色を生成してくれるだろう。しかし、それは感動的な音楽になるだろうか？実際の音楽家がそれを使うだろうか？もちろん使わないだろう。(ドナルド・ノーマン「インビジブル・コンピューター」101p)

本研究全体の大きな問題意識は、大きくまとめると、**コンピューターで音楽を作るための方法を自分自身で開拓することがなぜ難しいのか**とまとめることができる。

しかし無論、パーソナルコンピューターが普及して以降の音楽制作において、自らが音を生み出す方法を自らの手で作り上げる DIY 的アプローチは積極的に行われてはいた。本章では、まずこの 90〜00 年代の音楽制作におけるコンピューターの利用方法を振り返り、それらは自らの手でブラックボックス化された音楽にまつわるテクノロジーを、誤用などを積極的に用いて開拓する、カスコーンの “失敗の美学” に代表される態度であったがしかし、最終的には失敗することにすらも失敗した歴史であった、と位置付けることを試みる。そして 2020 年代に音楽家が取ることのできるアプローチとして、ブラックボックス化され、不可視となったインフラストラクチャを自らの手で開き、自らの手で作り上げ進んでいくという方法を、Human-Computer Interaction(HCI) におけるインターフェース研究、科学技術社会論 (STS) におけるインフラストラクチャ研究、メディア考古学、グリッチアートといった様々な領域を参照しながら提示する。

2.2 失敗することに失敗したソフトウェアとしての音楽

2.2.1 カスタマイズとは？

そもそも、コンピューターで音楽を作るための方法を自分自身で開拓する、というのはどういうことだろうか。2001 年に電子音楽家の Oval(Markus Popp) は” 音楽制作ソフトウェア環境は、普遍性という側面を持っていながらも、過去の音楽的遺産、伝統的な音楽の理解方法のコンテナとして機能しているにすぎない”(ポップ 2001) と発言している。パーソナルコンピューターは普及し、誰もが自分で音楽制作の環境をカスタマイズ

できるようになっても、録音音楽の慣習に基づいたワークフローのシミュレートから（時代に伴い緩やかな逸脱を見せはじめつつも）脱することはなかったし、2021 年においてもその状況は大きく変わっていない。

とはいえ、現代の音楽制作ソフトウェアにまったくもってカスタマイズ性がないかといえばそうではない。リバーブエフェクトが足りないと思えば、どこかのメーカーからリバーブエフェクトのプラグインを購入してきて、DAW ソフトウェアから読み込めばそのソフトウェアの機能は確実に拡張されている。しかしもっと根本的な機能ー例えば 5 拍子と 4 拍子が混ざったポリリズムの曲を作りたいので、この両方を別々のヘッドホンから鳴るようにしたいとなれば、それは正規の利用方法では無理だ。DAW ソフトウェアの中にはメトロノームは 1 つしかない。MIDI トラックに 4 拍子と 5 拍子の頭にノートをそれぞれ配置していけば望む機能は実現できなくはないが、そもそもメトロノームが 2 つとか 3 つとかに増やせればその方が手っ取り早いのは自明だ。

こうしたソフトウェアの拡張性の違いは、HCI 分野における End-User Development、つまりソフトウェアのユーザー自身がソフトウェアの機能を拡張していく方法についての研究において 2 段階に分類されている。

1. Parameterization or customization. Activities that allow users to choose among alternative behaviors (or presentations or interaction mechanisms) already available in the application. Adaptive systems are those where the customization happens automatically by the system in reaction to observation the user's behavior.
2. Program creation and modification. Activities that imply some modification, aiming at creating from scratch or modifying an existing software artifact. Examples of these approaches are: programming by example (also called programming by demonstration), visual programming, macros, and scripting languages(Lieberman et al. 2016).

つまり現代の音楽制作ソフトウェアには 1. のパラメーター化、カスタマイズはできても多くは 2. プログラムの機能そのものを変更することができないと言える。

2.2.2 パーソナル・ダイナミック・メディア

しかし、そもそもの自らの手で機能を拡張できる道具としてのコンピューターの思想で重視されていたのは 2. のプログラムの機能そのものを変更する思想だ。

コンピューターが単なる高速計算装置から、様々な表現の形態に対応可能な汎用入出力装置へと変化させた思想の直接的な源流は、Kay と Goldberg による"Personal Dynamic Media" に見ることができる。Kay らは Dynabook というあらゆる情報-ー詩、レシピ、レコード、絵、アニメーション、楽譜、波形や物理シミュレーションなどを本のように格納しを蓄積し、Smalltalk という対話的手法を得意とするプログラミング言語と組み合わせることで、テキストだけでなく音や画像といったあらゆる種別の入力を処理しリアルタイムでまた何かしらの形式でアウトプットするような**メタメディア**をデザインした。彼らのモックアップはちょうど今日のラップトップのような見た目だったが、当時のコンピューターの処理速度ではまだそのサイズでは実現不可能であったため、デスクトップコンピューターにディスプレイ、音楽キーボード、タイプライター、マウスなど多様な入出力デバイスを組み合わされた 1 つの勉強机のような構成で、様々なアプリケーションを作成して、中学生程度の年齢を含む幅広い対象に使用してもらう実験を行った (Kay and Goldberg 1977)。

この中で音楽関係のアプリケーションを抜き出すと、今日の楽譜編集ソフトのように、キーボード入力をキャプチャしてその後マウスなどで編集ができる楽譜編集アプリケーション OPUS、同様にキーボード入力の取得と再編集が可能な今日のいわゆるシーケンサーアプリケーションのような TWANG と呼ばれるシステム、のちに TWANG に統合したとされるシンセサイザーの音色編集のアプリケーションが挙げられている。

ただしここで注意して読むべきは、まずコンピューターを用いた音声の合成それ自体の取り組みは第 4 章で見るように 50 年代から行われてきていること、リアルタイムでの（それも専用のハードウェアに依存せずCPU 処理とデジタル-アナログコンバーターのみでの）音声合成はこの時代としては比較的目新しいと言える

ものの、その信号処理は任意にプログラムできるようなものではなく、ちょうどこの時期 Chowning によって提案された FM（周波数変調合成）と呼ばれる方式での合成をサンプリングレート 13.5kHz、量子化ビット数 12bit という精度で、同時に 5 ボイスまでを制御できるという、ある程度制限されてはいたものだった。

また、“Personal Dynamic Media” ではミュージシャンによってプログラムされた、と見出しが付けられてはいるものの、他の例えば花びらの模様を 14 歳の子が自分でプログラムしたという状況とはやや違い、TWANG をプログラムしたのはのちに Smalltalk の開発に中心的に関わり続けている Ted Kaehler だとされていることには留意したい。ただ、楽譜キャプチャシステム OPUS を作ったのは Chris Jeffers という、コンピューターサイエンティストではなく音楽家かつ教育者だったとされる人物で、OPUS では今日の類似アプリケーションでも珍しい、演奏したノートの強弱からテンポのゆらぎを推定した上で改めて音符を配置するという機能を持ち Kay たちを驚かせたという。(Kay 1993) 実際のところ、Jeffers のようなプログラミングを専門としない人のアプリケーション開発において、どれほど Kay や Goldberg による支援なしに自力でプログラミングが行えていたのかに関しては定かではない。とはいえひとまず Kay らの Dynabook において、ユーザーが可変的にメタメディアの機能をチューニングしていく過程では、与えられたパラメーター化だけでなく、Jeffers のような自らが楽譜キャプチャシステムという仕組みのモデル化を行うことで独自のシステムを作り出しており、そこには Smalltalk というプログラミング言語の存在が欠かせなかった、ということは言えるはずだ。メディア研究者の Manovich も Kay らの理想としての Universal Media Machine としてのコンピューターには、ユーザー自身がプログラムを組めることが重要な点であったと指摘し、しかし結局プログラミング可能という理想はインタラクティブ GUI を持つ商用パーソナルコンピューターとして現れた Macintosh などには引き継がれることはなかったという見方を示し、90 年代以降の Perl や PHP、Python、Javascript といったスクリプティングプログラミング言語の登場によってようやく、ユーザーが自分自身でソフトウェアを作る環境が登場してきたと述べる。そしてこうしたユーザーがプログラムを組無ことができるの中での、画像や音声といったマルチメディア処理の分野における代表としては、Reas と Fry による画像/映像表現のためのプログラミング環境 Processing や、音楽においては Puckette による Max、Puredata を挙げている (**Manovich2001**)。

2.2.3 「音楽のデジタル化」は生産と消費の関係をどう変えたか

それでは、当初の思想にも関わらずなぜ現代のコンピューターにおけるメディア処理は自らの手で可変しづらくなっているのか。それはパーソナルコンピューターが実際に実現して以降、ソフトウェア自体を商品として売り買いする経済的側面にあると考えられる。

音楽制作ソフトウェアを使う音楽家は経済的な側面から見れば、音楽を生産する者であると同時に、出来上がったソフトウェアやプラグインを消費する者（= Customer）という二重の存在としても見ることができる。音楽を作るソフトウェアを自らの手で作るべきだと考える数少ない人の中には、DIY 的に音楽を作る手段そのものを自らの手で作る理由に、音楽制作ソフトウェアの市場という大量消費社会のイデオロギーへの反発を持つものもいる。例えば Reaktor[*1]を用いた音楽ソフトウェアを自分で制作し音楽作りに積極的に使っている Squarepusher(Tom Jenkinson) は以下のように発言する。

> 僕が強く異議を唱えることのひとつに、「音楽作りのマーケット化」ってことがあってね。それはどういう意味かというと、今の時代、音楽を作る人びとは『この最新の商品を買わなくちゃ』って風に駆り立てられているわけだよね？　それはもちろん、ソフトウェア製造会社、あるいは楽器メーカーによるマーケティング・キャンペーンが拍車をかけているんだけど.....特に若いミュージシャンの場合がそ

[*1] reaktor の解説。Squarepusher は音楽制作にソフトウェアを用いていると度々発言してはいるが実際にどういったソフトウェア環境で制作をしているかの詳細についてはどのインタビューでも触れられておらずはっきりしない。2014 年？のインタビュー映像では Puredata、Max、Reaktor に言及しているが、映像として写されているのは Reaktor のみであった。https://www.youtube.com/watch?v=zEofbwt0zRY&t=534s

> うなんだけど、彼らはどういうわけか、新しい音楽を作り出す、今風で.....そうだね、そこにファッショナブルなって要素も加わるんだろうけど、そういったトレンディで新しい音楽を作るためには、この最新の楽器、あるいはこの最新のソフトウェアをゲットしなくちゃダメなんだ、そんな風に感じるように仕向けられている[*2]。

2000 年代初頭には、音楽にパーソナルコンピューティング、あるいはユビキタス・コンピューティングの思想が関わることで、音楽制作ソフトウェアに限らず、音楽における生産と消費の関係が変わるという議論がなされてきた。

例えば増田は Korg 社の KAOSSILATOR や YAMAHA の TENORI-ON のような 2000 年代に登場した音楽フレーズ自体を演奏として操作できる電子楽器を例に挙げ、それらの発音ボタンを押す行為は iPod のようなデジタル音楽プレーヤーというーどちらも物理的にはコンピューターによって音声処理をするデバイス[*3]の再生ボタンを押すと言う行為：Play にどれほどの差がつけられるのだろうか、という視点から、音楽がデジタル化することによって消費と生産の区別がつかなくなるという議論を展開した (**Masuda2008**)。

増田はこれら TENORI-ON のような楽器を「Play」する行為を Attali の作曲の系（レゾー）の到来となぞらえる。Attali は” 音楽は予言である” という、音楽の変化が社会の変化に先んじて起きると述べた著作「ノイズ（Bruits)」で、社会における音楽の位置付けを儀式、演奏、反復、作曲という 4 つの時代 (系) に分け、録音技術の登場により発生した、反復の系における「麻酔をかけられた市場のための、大量生産される音楽（182p〜183p)」に続き、作曲の系という「諸個人が、自分自身のために、意味、使用、それに交換を度外視して楽しむために生産する音楽」の時代（と、それに呼応する消費経済に替わる社会構造）が訪れる可能性を議論した (**Attali1985**)。

一方で篠田は音楽のデジタル化において重要な役割を果たした MIDI(電子楽器の相互利用のための演奏情報などを定義したプロトコル) 規格成立までの過程の調査において、カナダのポピュラー音楽学者ポール・テベルジュの議論を取り上げ増田と対比し、“「音楽のデジタル化」とは一方でそれ以前は特権的なものであった音楽の制作テクノロジーを消費者へと解放していく過程であり、他方で音楽の制作者であるミュージシャンを消費者の一類型として取り込んでいく過程” であったと指摘する (篠田 2018)。

> ミュージシャン達はプリセットされた新たなサウンドを求めて、様々なデジタル楽器を次から次へと買い足していくようになり、そのカタログやレビューメディアとしての役割をミュージシャン雑誌は果たしていくようになる。(中略) もはや、ミュージシャンは楽器をゼロから演奏して音楽制作に勤しむのではなく、デジタル楽器にプリセットされた音色やパターンを駆使して、音楽制作を行っていくようになったのであり、ミュージシャンはそのようなテクノロジーを使って音楽を生産する時に、同時にそのテクノロジーを消費し再生産する存在なのだ。テベルジュによれば、音楽制作のデジタル化とは、このように音楽の生産者たるミュージシャンがある種の消費者として再編成されていく過程であるのだ。

実際、Attali の反復の系における解説を読むと、現代の音楽ソフトウェア産業の状況にそのまま当てはまるようにも読める。

> 音楽は音楽家をすり抜ける。たとえ、音楽家が、構造を考え、それを、彼らの経験（あるいはむしろそれから離れて）をもちい、曲解された数学からのしばしば誤った借用物によって、理論化していると信じようとも、彼らの役割は、音の生産の予見不可能な展開を誘導するのが精一杯で、とても、しっかりした楽器から生じる予測できる音の組み合わせどころではない。「作曲家はボタンを押すだけのパイロットのようなものになる」。ほとんど有機的な、その歴史、その意義によって音楽自体を生み出す作品。作曲家が、作品のために発明された楽器を往々にしてもたないために、表象することのできないような作

[*2] https://www.ele-king.net/interviews/004425/index-2.php
[*3] のちに挙げる Norman が言う、“情報アプライアンス” と置き換えてもいいだろう。

> 品。経済学においてと同様、いや、それに先がけ、音楽は、人間の知識と道具による人間の凌駕の実験となる。もはや生産の発展は組織されず、ただ、良く知られていない法則（音の素材の法則）へと進化を調整しようと苦心するだけだ。生産したいと思うもののテクノロジーを作り出すかわりに、テクノロジーが可能にするものを生産する。

Attali が反復の系と作曲の系を分かつものとして議論の中心においていたのは、大量消費社会が産む、一方で個々の商品の差異によって価値を生み出しながらも、他方では生産物と消費者のスペクトラムがそれぞれ自らを画一するように収束し続けるという矛盾した傾向である。それゆえ音楽ソフトウェアやプラグインも録音音楽そのものの販売と同様の市場原理に基づいてブランディングやスタイル付けをされ続ける以上は、作曲の系に突入したと見せかけて反復の系に逆戻りという状況を産むのも必然と言える。

また Attali がここで「作曲家はボタンを押すだけのパイロットのようなものになる」と引用しているのは 20 世紀の作曲家 Iannis Xenakis の言葉だ。Xenakis は 20 世紀の作家の中でも数学、特に確率統計的アプローチによって音を操ることで新しい音楽を生み出す試みをしていた作曲家で、後年は UPIC という、二次元平面上のドローイングを直接的に音声へと変換するプログラムを CEMAmu(現 CCMIX) という研究所と共同で開発していた。Attali が録音音源による表現の画一化を厳しく批判する中で Xenakis のような次世代の楽器、あるいはより高度な音楽生成システム作り自体を行うとしていた Xenakis の発言を引き合いに出しているのは、単に音楽を作る（あるいはそれに準じた何かしらの制作行為）ことを皆がはじめればそれが作曲の系であると言うことではないことを端に示している。

> 作曲家は退屈な計算から解放されて，この新たな音楽の形が投げかける一般的な問題に没頭し，さらに入力データの値をいじるなどしてこの音楽形態の詳細な調査に専念することができる．たとえば，ソリストや室内オーケストラから大オーケストラまでのありとあらゆる楽器の組み合わせを試してみるのもよいだろう．電子頭脳という助っ人を得たことで，作曲家はいわば飛行士となる．ボタンを押し，座標を入力して，音の空間を航行する宇宙船を監督，制御し，以前は遠い夢でしかなく覗き見るだけだった音の星座や銀河を抜けて、さらに前に進むことができる．今や安楽椅子に座ったままで星座や銀河を自在に探検することが可能なのである．

2.2.4　無限のバージョニングとなめらかな忘却

メディア編集ソフトウェア環境において、ユーザーが自由にプログラムを作ることができない理由を経済的な観点から省みることはソフトウェアを研究するという行為そのものの根幹にも関わる。Manovich は 2013 年の「Software Takes Command[*4]」で、誰もがソフトウェアを使って表現を行っているにもかかわらず、ほとんどの人がその学術的な由来や文脈を知らないー例えば映画を作る人で Lumiere 兄弟を知らない人はほとんどいないのに対して、コンピューターを使って表現をしている人のうち何人が Alan Kay を知っているだろうかーという、メディア研究におけるソフトウェアに対する無関心に対する疑問を呈し、その大きな理由は経済的なものではないかと考察した。画像編集アプリケーションや CAD アプリケーション、音楽制作アプリケーションなどのソフトウェアが、映画や、（同じくソフトウェアである）ゲームタイトルと違って、数十年前の作品がリイシューされて販売されることなどほとんどない。というよりも、例えば 2000 年の Microsoft Word と 2021 年の Word とを並べて、20 年前のバージョンを好んで取り出してきて使ったり、Microsoft が 20 年前のバージョンをリイシューして販売したり、あるいはそれらの違いについて議論、批評する人は音楽や映画と違いそうそういないし、そもそも動作させることすら難しい。音楽プログラミング言語でも同様の例はたくさ

[*4] 日本語では未訳だが、“ソフトウェアが指揮を執る” と訳されることが多い (**Manovich2017**)。元の意味合いとしては、コンピューターソフトウェアがコマンドを受け付ける、ということと、文化の中でソフトウェアが支配的な役割を果たしつつあるという両義性を込めたものと思われる (“ソフトウェアが命令を司る”、といった方がニュアンスとしては近いように思える)。

んある。Cycling'74 Max のバージョン 4 と 5 の間でファイル保存の形式が大きく変わりインターフェースの見た目も変化した。これは、主にグラフィックの処理にこのバージョンから JUCE という C++ 言語向けフレームワークを導入したことが大きな理由であり、機能としても実装としてもバージョン 4 からバージョン 5 の間には大きな開きがある。何より、こうした 2 つのバージョンのソフトウェアは（単に機能としても、そしておそらくソフトウェアの実装的な面から見ても）もはや別物と思える程に機能が増えているけれども、表面上は同じソフトウェアだという連続性を持っている。

バージョン付けの規則としてプログラマの中で最もよく使われている Semantic Versioning という規則では、`v2.14.3` というような 3 つの数字の並びによってソフトウェアのバージョンを表現する。この時、下位の数字から順に、3 はバグ修正などが行われたときに増加するパッチアップデート、14 は前方互換性がなくなるー簡単にいえば小さくとも何かしらの新しい機能追加を行うマイナーアップデート、そして 2 は後方互換性の破壊、つまり、昔のバージョンで使えていた機能が使えなくなってしまう変更を行うメジャーアップデートを表す[*5]。ソフトウェアの長い積み重ねを 3 つの数字で表すバージョン番号で最もメジャーな数字はすなわちそのソフトウェアにおける見えない歴史的切断の回数を表している。

このように、ソフトウェアは新しい機能を日々追加されながら、時にメジャーアップデートなどで追加でユーザーから料金を取ったり、華々しく新機能についての宣伝で前バージョンとの差異を強調しながらも、同時に一方で両者は同じソフトウェアであるというアイデンティティを保ち続けており、それによって後から歴史を振り返ったときに文化的な系譜学（cultural genealogy）を編みにくくする。

2.2.5 「インビジブル・コンピューター」の思想

> 良い道具とは不可視な道具のことだ。不可視であることによって（そのツールがあなたの意識を邪魔しないという意味、で) あなたは道具ではなくタスクに集中することができる。メガネは良い道具だ– あなたは眼鏡を見ているのではなく世界の方を見てるのだから。盲人は杖を叩くことによって道の様子路感じることができるのであって、杖を感じているのではない。もちろん、ツールはそれ自体が不可視なわけではなく、使う文脈の一部分としてそうなるのだ。十分な練習を積めば、いろいろある一見難しそうなものでも見えなくすることができる。私の指は vi の編集コマンドを知っており、それは意識の中からはすでに長いこと忘れ去られている。しかし良い道具とは不可視性を高めるものだ。(ワイザー、“Thw World is Not a Desktop”)

ノーマンは「インビジブル・コンピューター」の序章にて、蓄音器 (フォノグラフ) を発明したエジソンがビジネスとしては失敗したことを例に挙げて、技術的に最良の選択が必ずしも市場的に成功するわけではないという説明をしている。エジソンのフォノグラフは円筒状の記録媒体を用いていたので大量に複製することが難しいものの、音質としてはのちにベルリナーが発明した円盤型のレコードよりも優れていたとされる。しかしノーマンはエジソンが技術的優位性に固執して、ベルリナーのレコードでも音質は当時の聴衆としては十分なもので、ユーザーが求めるものを正しく把握することによって

そしてノーマンはこのインビジブル・コンピューターの発展のためには標準規格というインフラストラクチャの重要性があることも意識していた。

> 問題は、蓄音器にしてもコンピュータにしても、テクノロジを変えるのは比較的優しいが、社会的、組織的、文化的な部分は変えるのが難しいということである。
>
> 一度インフラが確立されると、それを変えるのは困難である。新しい方法が優れた結果をもたらすことが明らかであっても、古いやり方がしぶとく残るものだ。それがあまりにも社会の文化に深く埋め込まれ、人々が生活や仕事や遊びで覚えたやり方に深く染み込んでいるために、変化は非常に遅く、時には

[*5] https://semver.org/lang/ja/

数十年もかかるからである。

しかしここで興味深いのは、ノーマンがこのようなインフストラクチャの影響力が大きい情報アプライアンスの中で「最も有効で、広く受け入れられている例 (73p)」として挙げるのが MIDI 規格なのである。

> 音楽アプライアンスは、全てのアプライアンス間の情報共有に関する国際標準規格として広く受け入れられた MIDI(Musical Instrument Digital Interdace) の出現によって実際に可能になった。MIDI のおかげで、どんな会社でも他のすべての音楽アプライアンスと自由にやりとりできる新しいアプライアンスを発明することができる。(中略) これにより音楽家は非常に多くの種類の楽器と音作りのための機器を組み合わせて、新しい曲を容易に作り出すことができるようになった。(73˜74p)

日本における MIDI 規格成立の過程を調査した篠田ミルによれば、MIDI 規格の仕様に賛同し制定に関与したメーカーを「勝者」に、そこから排除されたメーカーを「敗者」にすることにつながったという、デファクトスタンダード的、排他的性格を持っていたことも指摘されている。

> 情報アプライアンス業界の発展のために最も重要な教訓は、情報交換のためのオープンで汎用の標準を確立することの大切さであろう。われわれが情報を共有するための世界標準を打ち立てることができたときのみ、個々のアプライアンスごとに使われていた特殊なインフラは不要になる。それぞれのアプライアンスはそのニーズにぴったり合うものなら何でも使えるようになる。どの会社もその運営に最も良いインフラを選ぶことができる。情報交換が標準化されさえすれば、他のことは問題ではなくなるのである。(174p)

現在の視点から見てみれば、このようなノーマンの考え方は、インフラストラクチャを設計すること自体の困難さに自覚的である割には楽観的で理想化されたビジョンであったと言えるだろう。実際、ギャロウェイは TCP/IP や HTML といった今日の情報インフラストラクチャの中でも最も基幹的部分とも言えるプロトコルとフォーマットの成立過程を分析する中で、“インターネットプロトコルは、組織にかんする分散型のシステムを生み出す助けとなるのだが、その一方でそれら自体が非分散的で、官僚主義的な制度によって下支えされているのである (239p)” と述べるように

2.2.6　サーキットベンディング、失敗の美学、グリッチ

では、パラメーター化という開放された箱庭で遊ぶに過ぎない状態に対して、パーソナルコンピューター時代に音楽制作者はどのように単なる消費者に成り下がる潮流に抗ってきたのか。それを代表するのが、テクノロジーを戦略的に誤用する、カスコーンの「失敗の美学」に代表される態度だ。

増田の議論における”CD を再生することも、サンプラーを操ることも、人がテクノロジーを介してデジタル・データを操作し、音響へと変換している点では等しい” というある種の唯物論、もしくはマテリアリスム的価値観は、メディア論などでもコンピューターというメディウムならではの表現の特徴を考察する中では支配的なものである（例えばマノヴィッチが「ニューメディアの言語」でニューメディア≒コンピューターを用いた表現の特性の初めに「数値による表象 (Numerical Representation)」を挙げている (**Manovich2001**) ことなど)。

カスコーン 失敗の美学 テクノロジーを戦略的に誤用すること

市場原理からの逸脱

2.2.7　Oval とソフトウェアとしての音楽

Oval は CD の読み取りエラーを積極的に音楽に取り入れるような、グリッチと呼ばれる手法を用いる作曲家で、2000 年代に ovalprocess という自分のためのカスタム音楽制作ソフトウェアを制作し、ソフトウェア自

体を音楽作品（さらにそれを超えて、レーベルやエディションに相当するもの）にしてしまうことを試みた。彼の ovalprocess に関する論考で興味深いのは、テクノロジーが新しい音楽を生み出すと言った技術決定論的な言説に限らず、規格やフォーマットといった制度論に対して強く自覚的になっている点だ。いくつか引用しよう（いずれも強調は筆者による）。

> オヴァルは、デジタル制作メディアにおける、一つの典型的なワークフローの**スタンダード**を、攻撃的に使用する。
>
> ～私の音楽は、改定された「音楽 Ver2.0」という、新しい決定的な**スタンダード**を導入しようとしている。
>
> 音楽制作ソフトには潜在的なリベラル的傾向があるにもかかわらず、私はその「クリエイティヴ」な可能性ではなく、ワークフローの大規模な**規格化**に焦点を当てている。
>
> 私の個人的アプローチは、**スタンダード**を攻撃的（オフェンシヴ）に使うことに基づいている。デジタル（音楽）メディアを「実験的」に使用するよりも、スタンダードーーファイル・フォーマット、ファイル・トランスファー・プロトコル、圧縮スタンダード、コーデック（データの圧縮／慎重、COmpression ／ DECompression の略）、オペレーティング・システムーーの方が、私の作業過程において重要な要素なのである。それは、これらのスタンダードが、何がコンピューターで可能であるかを決定する**土台**であるからだ。
>
> 音楽制作ワークフローの論議に対する大衆の認識と接し方が変わらなければ、「独自」のソフトウェアを作る利点は疑わしく、何も解決しないだろう。電子音楽制作の根底にある**技術的スタンダード**に関する論議を、徹底的に見直す必要がある。カスタム。ソフトウェアを作ったとしても、それを廃れた概念や、時代遅れの音楽パラダイムに従ってデザインし、使用し続けたのでは、何の解決にもならないのだ。

こうした議論を見ると、Oval がグリッチのようなアプローチを取っていたのは、ツールを意図的に誤用したり、アマチュアリズム的に「あちこちいじくり回す」(カスコーン 2005) ことによって隠された内部構造をあらわにしたり、無目的に聞いたことのないノイズ音を生み出したりすると言った目的でも、コンピューターのメディウム固有性としてマノヴィッチが挙げる、あらゆるタイプのデータを数値として表象する (Manovich 1999, p8) ことを逆手にとって、データを意図的に誤った読み替えをして新しい音楽を作るということに着目していたわけでもない。

むしろ、全てが数値として表象されるときに、見えないところで大企業や標準化団体がその意図を直接的には見えない形で埋め込み、創作者に対して間接的に影響を与えていることへのカウンター行為としての積極的な規格からの逸脱や、オルタナティヴな野良規格の構築という行為だったと見ることができるだろう。

最終的に ovalprocess はいくつかのインスタレーション展示を除いて公にリリースされることはなかった (引用)。

2.2.8 Ovalprocess はなぜ普及しなかったかー野良スタンダードの困難さ

まとめれば、音楽において情報インフラストラクチャを作るということはその時点での音楽を構成する要素の概念モデルを定義し、その概念モデルを境界線としてユーザーとデベロッパーを切断するための仕組みだといえる。

この切断によって、ユーザーは概念モデルより下層の構造の内容について意識する必要がなくなり、認知的な負荷を下げ、よりその人の集中したいタスクに専念できる。問題は、この概念モデルはある一定数以上の人数について共有されてなければ意味をなさないことと、概念モデルを標準規格 (例えば MIDI) やプロトコルによって定義するためのコストを誰が払えばよいのかということである。

2.2.9　iPhone 音楽アプリの衰勢：いつか音楽と呼ばれる（はずだった）もの

RjDj
Audible Realities

- なぜうまく行かなかったのだろう？仮説を立てることしかできないが、制作に対するインフラストラクチャ（MIDI など）に対しては意識的だったが、流通、受容に関するインフラストラクチャに対する意識が希薄だったのではないだろうか。
  - 徳井の iPhone 音楽アプリは機能がシンプルすぎて App Store の審査には通らなくなってしまう（メディアアートの輪廻転生）

2.3　後期ノーマンの主張：

複雑なものはそもそも複雑なんだからしょうがない的な

> デザイナーの仕事は、人々に適切な概念モデルを与えることである。(40p)
>
> 人の活動には、その痕跡を残すという副次的作用がある。これはほとんど意識的な気づき無しに行われるが、この服じてき作用は重要な社会的サイン (松浦:SIGN であることが重要) である。私はこれを「社会的シグニファイア」と呼ぶ。
>
> シグニファイアは、それが意図的なものであれ、意図的ではないものであれ、適切な行動への知覚可能なサインである。

2.3.1　仮説：静的可変性と動的可変性の両立

今日のコンピューターの姿の祖とも言える、ケイらによって提示された Dynabook のような、1 つの機器の機能を目的に応じてユーザーが変化させていくようなコンピューターのあり方と、ノーマンが情報アプライアンスと呼ぶ、機能を絞ることでユーザーの認知的負荷を下げるコンピューターのあり方を対比すると、**コンピューターの可変性を動的なものとして扱うか、静的なものとして扱うか**という違いと言い替えることができる。

コンピューターはプログラミングという行為によって任意に機能をカスタマイズすることができる、実質的に万能な装置ということにはどちらも変わりないが、その可変性を発揮するタイミングはいつなのかということである。しかし、ここで**誰がその機能を変化させるのか**という視点を考えると、この二つの思想は何も二項対立ではないことがわかる。

つまり、売られているときの機能としては単純な情報アプライアンスであっても、ユーザーが望めば自分の手でその機能をカスタマイズできるようなコンピューターのような機器は考えられないだろうか？

2.4　ブラックボックスとしての音楽インフラストラクチャを分解する

ここでいうブラックボックス性とは、元来サイバネティクスの中で立ち上がった言語化 (翻訳) 不可能性という意味に近しいものだったが、現在では主に次のような意味を複合したものとして使用される。

- 理解不可能性 (Inunderstandability)
- 知覚不可能性 (Inperceptibility)

- 翻訳不可能性（Intranslatability）
- アクセス不可能性 (Inaccessibility)

2.5 Koskinen による構成的デザインリサーチの 3 つのフィールド

Koskinen は RtD を実践の発生する場所として Lab、Field、Showroom の 3 つに分け整理した。Lab における RtD は本論文でも中心的に議論の対象となっている、ソフトウェア、インターフェース、特に NIME 研究にも代表されるもので、生態学、認知心理学などに由来し、プロトタイピングと評価実験を繰り返しその価値を提示する。Field での RtD は文化人類学やエスノグラフィに由来する、ユーザーがいる環境を社会科学的な質的な調査法を中心に理解することを目指した研究である。そして Showroom での RtD はいわゆるクリティカル・デザイン (**Marpus2019**) やスペキュラティブ・デザイン (**Dan2013**) に代表される、展示や映像といった形態で、必ずしも現代において役立つものではない人工物を提示することを通して、社会構造に対する批評的役割を担ったり、議論を促すことを目的としたアプローチだ。

特にクリティカル・デザインは Papanek の「生きのびるためのデザイン」(**Papanek1971**) に代表される、大量消費社会における功利主義的なプロダクトデザイナーに対する批判を受け継いだものでもあり、本研究とモチベーションを多く共有するところがある。だが Showroom での提示は方法論としては、現場とは異なる場所で議論を引き起こす以上、常に実際の問題への直接的な議論からは遠ざかりかねない危険性もある。Marpus は” 実証主義的なリサーチは知識を生み出すたびに問いかけを止めてしまう”(**Marpus2019**) ことによりデザインという行為における批評性の重要性を強調するが、一方で現状のクリティカルデザインはその発表される場の性質として、芸術作品と同じような枠組みでの議論に巻き込まれてしまい、デザインの実践における狙いを達成しきれない危険性も指摘し、そのためにはなるべく広いコミュニティを巻き込んでいく必要性があると言う。

2.5.1 トーマス・トウェイツ「トースター・プロジェクト」

このクリティカル・デザインの困難を説明しやすい例として、本研究のアプローチでもある、高度にブラックボックス化されすぎたテクノロジーを開く試みという共通点を持つ、トーマス・トウェイツによる「Toaster Project」が挙げられる (**Tweiz2009**)。トウェイツは彼が「近代の消費文化の象徴」と考える、トースターという工業製品が出来上がるまでに詰め込まれている人類の知識の蓄積を、ゼロからー例えば筐体を構成する鉄を、外装を構成するプラスチックを、電熱線として機能するニッケルのワイヤーを、すべてを自らの手で作り上げることで目の前に出現させる。

しかし、そのトースターのデモは結局、作った機械から煙が出ることでパンは焼けずに終了する。さらに作る過程では実は鉄を生成するためには電子レンジを（アブノーマルな使い方であるとはいえ）用いているし、プラスチックの原材料には廃棄された工業製品を（ある種の人間が捨てたものももはや自然の一部であるという人新世的な問題意識と重ね合わせることによって）用いているし、ニッケルワイヤーは eBay で売られていた記念硬貨を溶かして作られると言ったように、高度に洗練された工業製品を自らの手で作るにあたって、その工業製品溢れる文化の結果存在している材料や道具に依存するという、一見矛盾した状況を、手間やコストの理由から半ば強制的に受け入れることになっている。また何より彼は、トースターを作るとなった時に、では火でパンをあぶる事はトースターで焼いたことになるのだろうか？というトースターの存在論を突きつけられることになる。

久保田はこの Toaster Project を、“DIY を突き詰めていくと、それはやがてラディカリズムに至る。ラディカルとは、根本的、徹底的という意味であり、ラディカリズムとは、今のやり方を根本から見直すための行動でもある。” という、ラディカルな DIY として評価しつつも、同時に、“ある一時の熱中によって、半ば奇跡

的に現れたとしても、すぐに消え去ってしまうような儚いものである。” という一過性にも触れている (久保田 2020)。

実際、トウェイツのプロジェクトは、後から映像や書籍のようなドキュメント（Koskinen のいう Showroom の広義の形とみなせるだろう）として記録されることを前提としたパフォーマティヴな実践という側面もある[*6]。トウェイツのトースターやその制作過程を見ることを通じて、確かに私たちはトースターの背後にある技術の蓄積を実感はできるかもしれないが、それを見た人がトースターを自らの手で作れるようになるわけではない（むしろ遠ざかってしまうかもしれない）し、未来の工業製品の作られ方にどれほどインパクトを与えられているかは疑わしくもある。

例えばこのような” 一見高度そうなテクノロジーを自らの手でゼロから作ってみる” というプロジェクトはもはやデザイナーを自覚する者に限らずとも多数実行されている。例えば Youtube での動画投稿を中心に活動するメイカーの Sam Zeloof は、これまでパーソナル・ファブリケーションの動向が加速しながらも、個人が作ることはほぼ不可能だと思われていた集積回路 (IC) を、ガレージ程度の規模で実現する方法を追求し、1200 個のトランジスタを埋め込むチップを作り上げてしまっている。同様に、日本のメイカーであるゆなでぃじっく、またアメリカの Dalibor Farny はすでに商用製品としては時代遅れとされ生産されなくなった、冷陰極管の仕組みを用いて数字や文字を表示するデバイスであるニキシー管を、個人で作る方法をそれぞれ継続的に研究し安定した生産を行えるまでに技術を高めており、その様子を Youtube に投稿している。

Zaloof、でぃじっく、Farny の取り組みは共にデザインとしての取り組みというよりも、個人のエンジニアリング的興味を突き詰めた結果に過ぎないが、彼らがブログに残した足跡を辿ることで（金銭、設備のコストを度外視すれば）ある程度は再現することも可能になっている。彼らの残した動画やブログ記事はトウェイツが提示するのと同様の（あるいはそれ以上に）、大量消費社会とそれに伴うトップダウンに時代遅れなものと決め付けられるテクノロジー、ブラックボックス化されたテクノロジーを自らの手に取り戻せるという強いメッセージを発している。そこに存在するのはもはやパフォーマンスではない現実そのもので、What-If を提示して議論を引き起こしているヒマがあったらその If の中へ突き進んでしまえばいいという説得力がある。

2.6　ハッキングとデザインを名乗らないデザイン

2.6.1　サーキット・ベンディング

Garnet & Hertz

Nick Collins

> サーキットベンディングは変わりました。Reed Ghazala がこの単語を作った頃よりも、オモチャ内部の実装密度が高くなり、機能が一つのチップに集約されるようになったことが要因のひとつでしょう。90 年代までは、音を鳴らす、ライトを点滅させる、スイッチを読み取るといった機能ごとに個別の IC が使われていたので、IC 間の配線をいじくりまわす喜びがあったのです。今日のおもちゃは単一の邪悪な黒い物体に全てがコントロールされていて、再配線可能な接続箇所が 1 つも見つからないことがあります。（中略）今やサーキットベンダーたちは改造可能な中古オモチャを求めて、リサイクルショップや eBay を物色してまわる運命です。

[*6] なお、トウェイツはこのプロジェクトの後に、「GoatMan Project」という、自らがヤギと同じような歩き方ができる歩行具を作り実際に生活する、よりパフォーマティヴなプロジェクトを行ってもいる。

2.6.2 UCNV 「The Art of PNG Glitch」

UCNV による「The Art of PNG Glitch」[*7]は画像データの可逆圧縮フォーマットの 1 つである PNG ファイルをグリッチ (データを意図的に破壊することで意図しない画像や、音声の変化を引き起こす技法) させる技法をまとめた技術文書である。画像のグリッチは、最も簡単な方法では、バイナリデータをそのまま編集できるエディタで開き、適当な部分のデータを書き換えたり、入れ替えたり削除したりすることで成される。しかしこの方法は、JPEG フォーマットでは機能する一方で PNG フォーマットでは機能しない。PNG にはチェックサムと呼ばれる画像データが壊れているかどうかを判定するためのフラグが含まれており、適当にデータを壊すとこのチェックサムの機能によってシステムからは「画像が壊れている」と判断され表示することができなくなる。そこで UCNV はグリッチさせた後のデータに改めてチェックサムを付加するような、PNG グリッチ専用の Ruby 言語で書かれたライブラリを作成し、そのライブラリを用いた PNG フォーマットが生み出す多様なグリッチのバリエーションを例示したドキュメントを作成したのだ。

UCNV によればこの文書は「技術文書を模した芸術作品という意図で作成された」ものだという (**ucnv2020**)。彼の論考「グリッチアート試論」では、「グリッチ」と「グリッチアート」を区別し、半ば偶発的に発生し、発見された前者が、プログラマたちにより再現可能、永続可能であることが示されることによって、「再現性をもった事故」へと変質され、その一見矛盾めいた状態が後者を支えるものになるという。Rosa Menkman、Antonio Roberts、Benjamin Berg といったグリッチを扱う作家が、グリッチという事故を単に表象として扱うだけではなく、それに再現性を持たせる制作プロセスの理解を通じてはじめて作品の受容が達成されるという態度を指摘している。

“グリッチアートが時代固有の現象であるとしたらその固有性こそがグリッチアートの限界となる”

2.6.3 Hundred Rabbits

Hundred Rabbits(以下、100R) はイラストレーター、ライターの Rek Bellum とミュージシャン、プログラマーの Devine Lu Linvega の 2 人によるユニットである。彼/彼女らは帆船で (日本を含む) 世界中を旅しながら様々なソフトウェアを作り続けている。

彼/彼女らは ORC Λという生成的音楽シーケンスを作るための難解プログラミング言語を 2018 年にリリースしている (**Orca2018**)。この言語は Befunge という難解プログラミング言語（わざと読んだり理解することが難しくなるように作られた言語）にインスパイアされている[*8]。Befunge とは、チューリングマシンのような 1 次元のテープ状のメモリからデータや命令を読み込み、実行し、テープに書き戻すというような仮想機械を 2 次元平面に拡張した、チューリング完全[*9]なプログラミング言語で、ORC Λも 2 次元のグリッドにデータと命令を配置してグラフィカルに、かつ複雑な音楽のシーケンスを作ることができる。

ORC Λをはじめとした 100R のソフトウェアは、彼/彼女らがもともと Web 関連の仕事をしていたこともあり、Electron[*10]という、Web ブラウザ状で動作するアプリケーションを各 OS 状で動作するスタンドアロンアプリケーションとして動作させることが可能なプラットフォームを用いて構築されていた。しかし Electron は配布の簡単さに対して、最小限に部品を削ってはいるものの Web ブラウザを丸ごとアプリケーションに格納するような方式になるため、アプリケーションのデータサイズやエネルギー消費が必然的に大きくなる。一方世界中を帆船で旅する 100R にとって、（例えばニュージーランドで大都市近郊に行けたとしても）電源は

[*7] The Art of X という表現は、例えば慣用句的に技術やコツといった意味を持つ（例えば The Art of War と言えば兵法、戦術といった意味合いを持つ)。当然、この慣用句的意味合いと芸術という二重の意味が込められていると考えられる。

[*8] https://esolangs.org/wiki/Befunge

[*9] ある機械がチューリングマシンと同等の計算能力を持つ≒メモリが無限にあればあらゆる種類の計算がその言語状で記述できるという状態をチューリング完全と呼ぶ（出典？）

[*10] https://www.electronjs.org/

ソーラー発電のバッテリーに大きく頼っており、ソフトウェアのエネルギー消費は実用上の問題となっていた。またポリネシアのようなインターネットの回線速度が限られる中で XCode や Photoshop のようなプロフェッショナル向けソフトウェアのアップデートが 10GB のようなデータサイズでやってくることも問題となった。そして彼/彼女らはこのようなコンピューター自体が大量消費文化を前提としてしまっていること自体を見直すべく、自らの使うコンピューターを RaspberryPi[*11]のようなオープンソースかつ低消費電力なデバイスに切り替え、ORC Λをはじめとした彼らが作ってきたソフトウェアや、彼らの Web サイトを生成するためのプログラムまでを可能な限り単純なエコシステムで動作させるべく、C 言語で書き直すことをはじめた (HundredRabbits 2021)。

そして 100R はのちに、Nintendo Entertainment System(NES[*12]) に用いられた 6502 という CPU 向けのアセンブリ言語を勉強し、Uxn と呼ばれる最小限の要素 (仮想機械、アセンブリ言語、オーディオやグラフィックスなどの各種ドライバ) で構成された独自のソフトウェアエコシステムを作りはじめている。もともとは自分たちのソフトウェアを自分たちでホストするためのプラットフォーム作りではあったが、この仕様をオープンソースとして公開した結果、Web ブラウザをはじめ、ESP32 というモダンなマイクロコントローラ (https://github.com/max22-/uxn-esp32) から、Nintendo DS やゲームボーイアドバンスのようなレトロゲーム機 (https://github.com/asiekierka/uxnds、https://git.badd10de.dev/uxngba) の上で動作させるバックエンドが有志によって作られはじめている。

100R の活動は帆船の旅による (かなり字義通りの意味での) ノマディックな生活と硬く結びついた実体験から来る、技術を誰が Control(管理＝制御) するのかという問いや、技術の副作用としての環境負荷の問題を、自らのために、自ら技術を理解し、自ら実装することで異なる技術社会のあり方を実践している。その上で、その姿勢やプロセスを公開することで結果的に多くの人間を巻き込みはじめてもいる。この点では、深く隠蔽された技術を自ら実装することによって理解し、異なる可能性を提示しているトウェイツの実践から、実際に機能し、大企業の作るソフトウェアにはない価値を提示しユーザー（あるいは協力者）を巻き込んでいる 100R の実践はさらに一歩踏み出したものだと言えるだろう[*13]。

2.6.4　アマチュアリズム的ハックティビズムの限界

Hacking はブラックボックス 1,2,3 に対しては機能するかもしれないが、4 に対しては効かないだろう

ブラックボックスを構成する要素をこのように 4 つに分類してみると、カウンターカルチャーとしてのハッキングはもはや有効な手段とは言えなくなってくる。テクノロジーを誤用したから、だからなんだっていうんだよ？と。高度化しすぎた知識を、誤用することによって表に出すことには一定の効果があった。サーキットベンディングで音声合成 IC をめちゃくちゃに配線することでおかしな音を出したり、音声ファイルのデータをおかしな順番で読み込むことによってめちゃくちゃな音を出すグリッチも、その背後にある音声合成 IC の構成や、音声データフォーマットのテクスチャ/質感ともいえる物を表に引っ張り出してきた。

ハッキングが無効になる過程としてのひとつめのカテゴリとしては、アマチュアリズムが無効化されることだ。音が出る電子回路は今や音を出す以外の機能も全て一つの CPU の中に納められており、回路を適当に繋

[*11] Raspberry Pi 財団が中心となり発売されている、オープンソース・ハードウェアの低価格なシングルボードコンピュータ。もともとはコンピューターを用いた教育目的で立ち上がったプロジェクトだが、現在は産業、商業プロダクトのメインボードとして利用されることも珍しくない。音楽の分野では、monome の norns(https://monome.org/docs/norns/) や Critter&Guitari の Organelle のようなプログラマブル電子楽器であったり、電子音楽家の Go Hiyama を中心とするエコーズブレスの AISO という音楽家が作る生成的 BGM のプラットフォーム (https://aiso.ooo/) に使われている。

[*12] 任天堂が日本国外で発売した、ファミリーコンピュータ（ファミコン）のマイナーアップデート版とも言えるゲーム機。海外においてファミコンのエミュレーター（コンピューターソフトウェアとしてゲーム機のハードウェアを再現するもの）としてはこの NES のエミュレーターが主流である。

[*13] なお彼らの活動がアカデミックな研究から参照されることはこれまで少なかったが、電子楽器や音楽のインターフェースに関する国際会議 New Interfaces for Musical Expression(NIME) の 2022 年のキーノートに 100R が招かれる予定になっており、緩やかにだが接続を見せはじめている。https://nime2022.org/

ぎ変えたところで、機械そのものが動作しなくなっておしまいだ。データをグリッチさせようとして適当にビットを弄くり回しても、そこにはエラー訂正の仕組みが入り込み、データは再生不可能な壊れた (corrupted) ものとして扱われておしまいだ。その上でテクノロジーを誤用し背景にあるものを引き出そうとすれば、ハッカーもまた専門的な知識を頭に入れた上で、敢えて誤用するという態度が必要になってくる。フータモが言うところの”Tinkerer” から”Thinkerer” への変化だ。グリッチの例を取れば、UCNV の The Art of PNG Glitch(**UCNV2015**) の例を考えることがわかりやすいだろう。PNG ファイルは今使われる画像ファイルの中でも JPG と並んでスタンダードなもののひとつだが、先述したような、フォーマット自体にエラー訂正の仕組みが備えられているフォーマットである。UCNV はそのエラーを避ける形でなお PNG のフォーマットの持つ圧縮アルゴリズムを露呈させるようなグリッチを、自ら pngglitch という ruby ライブラリを作ることで可能にした。

さらなる無効化の過程は、ハックするのに時間がかかるようになることだ。PNG グリッチの例を考えれば、PNG フォーマットに対する知識をつけた（= 1: 学習によるブラックボックスを開けた）ところで、JPG ファイルをバイナリエディタで開いて適当にデータを壊すことよりも、ruby を用いたプログラミングによって PNG ファイルをグリッチさせることには、(例えライブラリの開発を自分でやっていなかったとしても) 時間がかかる。そもそもソフトウェアのハッキングはハードウェアと比べると基本的には

最後の無効化の過程が 4 の社会的都合によってハックが不可能になることだ。キットラーの CPU のプロテクト・モードリングプロテクション：モダンな CPU と OS の権限システム

音楽における例：プリエンプティブ・スケジューリング

ユーザープログラムは OS がタイムスライスする都合に関与することができない
xenomai の様なカーネル拡張を利用すれば関与できる（Bela）ーしかし、問題はほとんどのコンピューターの OS は自力で改造することができないことだ.

なるべく公平に多数のユーザプログラムが CPU を使用する時間を分配するので、ユーザープログラムから仮にいじれる様なオプションがあったとしても、結局はスケジューリング優先順位の奪い合いになってしまう。これは公共性をめぐる政治そのものである。

この状況下においてはもはやテクノロジーの抜け穴を突いていくセキュリティの vulnerability を突いていく様な Cracking 行為しか成立しないし、そうした獣道を作ったり見つける様な隙すらないほどに現代のコンピューターをめぐるインフラは舗装されすぎてしまっている。カスコーンの言葉を借りるならば、「失敗の美学」のような失敗をすることすら私たちにはもはや難しい。

さらに抜け穴を探す作業すらも往年のプロフェショナルなプログラマーが手作業でセキュリティホールを見つけ出す様なやり方から Fuzzing の様にそれすらもコンピューターで抜け穴を機械的に探し続けては埋める様な作業に移行しつつある。ハッキングという質の問題がコンピューティングリソースという量（あるいは、資本）の問題へと転化されていく過程は機械学習における計算リソースの問題をはじめとしてあらゆるところで発生している。

2.7 暫定的解決策：作り方の作り方を作る

- https://www.mollymielke.com/cc
- 橋本麦 Glisp Purpose-Agnostic Tools
- 8bit MixTape
-

2.8　小括

本章ではまず 2000 年代を中心に音楽のデジタル化によって創造（生産）と消費の二分構造が変化するという楽観論を紹介した。しかしそこには同時に、これまで経済的な枠組みとは無関係であった創造の活動さえも新たな消費へと回収されてしまう危うさを同時に抱えていた。

その具体例として、2000 年代に行われた、ovalprocess、RjDj を例とするソフトウェアとしての音楽作品の衰勢について検討した。これらソフトウェアとしての音楽は、パーソナルコンピューティングの性能向上に伴い、リアルタイム音声処理のプログラミングが可能になったことが 1 つの背景だったが、同時にシンセサイザーの普及に伴って立ち上がった、MIDI という既存の音楽様式の上に成り立つプロトコル（インターフェース）への反発が同時に存在していた。

これらプログラミングという行為を通して作られるソフトウェアとしての音楽とは、ケイとゴールドバーグによって打ち立てられた、自らに必要な機能をユーザー自身がプログラミングという行為を通してコンピューターの機能を変化させていく、Personal Dynamic Media という思想に源流を辿ることができるが、ケイらが構築した Dynabook の音楽への応用例を詳細に検討すれば、それは依然既存の音楽的様式の上に乗るものでしかなく、

この既存の様式を前提にしてしまう作用は、のちのワイザーやノーマンのインビジブル（不可視な）コンピューターという思想によって強調されてゆく。コンピューターの可変性を活用しながらも、人間の認知的負荷を下げるためある目的に特化された機器：情報アプライアンス

このインビジブルなコンピューターの思想は、ユーザー個人の使用体験という面から見ればその人の認知的負荷を下げることに一役買ったかもしれないが、それは同時に社会的な意味での創造/生産の過程を消費の過程へと転化するものだと考えることができる。

認知的負荷を下げるための不可視性は、内部の仕組みを理解せずとも、入出力の対応関係をすでに他の人が構築してくれた概念モデル＝本来のサイバネティクス的な意味合いとしてのブラックボックスとして信頼することで成立する。

しかしこのブラックボックスは時に自らの手で修復、改変、改造することができないアクセス不可能性として現れる。

この不可視なブラックボックスの性質は整理すれば、アクセス不可能性、知覚不可能性、翻訳不可能性、理解不可能性という 4 つに分類できた。

- とりあえず、音楽のインフラストラクチャの異なるあり方を考えることには意義がある。コンピューターと音楽表現の関係性を考えていく上で、DMI のような、新しい道具を作ることによる新しい表現というような個に閉じた取り組みよりも、人と人との間を暗黙的に結びつける、失敗の出来なさを誘発するインフラストラクチャの政治的作用について考える必要がある。
- Open Question: 現在において、生産者/クリエーターを消費者に転化させてしまうような箱庭を作るのではない形での、アタリがいうところの作曲のレゾーを実現できるような音楽情報インフラストラクチャの設計は可能だろうか？
  - →少なくとも、後述する音楽言語における Multi-Language Paradigm はすでに存在する表現の様式を言語仕様に埋め込むことで、その様式それ自体へのアクセスを結果として遮断することになる。Papert の言う High-Ceiling,Low Floor の思想に反するものだ。

第 3 章

How: 音楽プログラミング言語をどう研究するか

3.1　音楽プログラミング言語をどう研究するか

前章では、音楽プログラミング言語におけるブラックボックスを減らしていくことで、より音楽プログラミングにおいてユーザーが自由にプログラムを組めるようになり、長期的には、一度は失敗したように見える、デジタルした音楽における消費と生産の二分化からの脱却に改めてアプローチするきっかけになるかもしれないことを音楽のインフラストラクチャという視点から考察した。

ではそのような音楽プログラミング言語を設計するという行為は学術的にはどのように正当化できるだろうか、という問いを本章では検討する。

新しい音楽プログラミング言語や、コンピューターを用いた音楽の表現を科学的な研究として提示するには直感的には 2 つのアプローチがあるように見える。

1 つは、これまでできたことをさらに効率よくできるようになったと主張すること、もう 1 つは、これまでできなかった表現ができるようになったと主張することだ。合わせ技として、例えば人間にはできないレベルの複雑さのアルゴリズミックコンポジションをコンピューターの計算能力を借りることによって実現できるようにする、効率により新しい表現にアプローチするというものも考えられる。

さてこの時、できなかったことをできるようにしたタイプの研究の場合、新しくできるようになったことの何がよいのかに関しての正当化を求めるべきか、仮にそうだとしたらどのようにすればいいのかという問題がある。研究者とその音楽の表現を必要としている者（≒ユーザー）が分かれている場合にはどんなにニッチな表現であっても表現の多様性を押し広げるものとして正当化でき、一方で研究者自身が表現の当事者でもある時には、その表現ができたからどうした？といった構造があるように見える・・・。（もうちょっと整理したい）

こうした、音楽のためのソフトウェア制作を研究とする場合の位置づけとして、**デザイン実践を通じた研究（Research through Design:RtD）**と呼ばれるアプローチが考えられる。RtD とは簡単には、デザインという行為や対象が、単なるプロダクトからソフトウェア、インターフェース、サービスなど多様化する中、それを学術研究として捉えた時には、単に問題解決、仮説検証を行う科学とは異なる分野の学問として考える必要があるのではないかといった問題意識の中行われている様々な議論を、“不明瞭かつ個別固有の社会・技術的問題を対象とする臨床的、生成的研究”(**Mizuno2017**) として位置付けるような運動である。

もっとも、RtD 音楽のためのソフトウェア研究や新しい音表現のためのインターフェース（New Interfaces for Expression:NIME）研究においては、それらを包括する Human-Computer Interaction(HCI) の分野において、単に仮説を検証し量的な評価を行うのでは正当化することができないような研究をどう評価していくべきかといった意味でのディスカッションとしても扱われている言葉ではある。本章ではまずデザイン学全体における RtD の議論の変遷を水野大二郎の文献 (**Mizuno2014**; **Mizuno2017**) を中心に参照しつつ、音楽プログラミング言語、NIME 研究の中で RtD 的研究についての位置付けを行った論文との議論を並列すること

で、音楽プログラミング言語研究において、これまでより広範な意味での RtD、特に、長い時間をかけて社会構造の変革を促すことを目指す Transition Design のようなアプローチの研究の可能性を議論する。

3.2　RtD の変遷

HCI 研究の中での RtD の位置付けに大きな影響を与えた William Gaver の「What Should We Expect from Research through Design?」(デザインを通じた研究に何を期待するべきか？) という論文では、RtD には、Popper の言う科学的アプローチたる必須条件、反証可能性 (Falsifiability) がないことにより、RtD を科学から明確に区別する (**Gaver2012**)。デザインが今日対象とする問題は Rittel と Webber が 60 年代に導入し、90 年代に Buchanan がデザインの理論の中に位置付けた" 意地悪な問題 (Wicked Problem)"(**Buchanan1992**)—全ての問題やその解決のための試行が個別に異なりその試みを実証主義的な方法で理論として一般化することができないような問題だからである。

Gaver はその上でデザイン学における理論とは、作られた人工物に対する注釈 (annotation) 付きのポートフォリオのようなものだという提案をする。建築家 Christpher Alexander が提示し、2000 年代にコンピュータープログラムやインターフェースのデザインのための指標となったデザインパターン (**AlexanderDesignPattern**) のような、繰り返される試行から帰納的に得られる規則ではなく、個々の事例の個別性を保ったままに、自らが新しく作ったものの類似点や差異を見ることで、各事例の究極的個別性 (Ultimate Particular) を保ちながらも、拡張可能かつ検証可能 (Verifiable) な理論を構築できるというのだ。それゆえ、"デザインの知識は生み出した人工物に近づき留まった時最も信頼に足る"(**Gaver2014**) であるとする。

確かに、世界の中から未知の事実を発見する科学という行為と、世界の中に未知の事物を作り出しそれを解釈したりするデザインという行為はこの見方では全く別の体系に基づく学問ということとなる。

しかし、この議論に乗っかるのだとすれば、筆者がこれから行おうと思っている音楽プログラミング言語の特徴や評価の語彙を整理したりする一般化や抽象化は無粋なものとなってしまうように思える。そこで、ここからは RtD の歴史的流れを継続してみていきながらも、一般論だけでなく音楽プログラミング言語という分野特有の事情とも対比しながらより本研究の位置付けを詳細に検討しよう。

3.3　音楽ソフトウェア・インターフェース研究における RtD

3.3.1　Dahl による RtD としての NIME 研究の分析

Dahl は音楽のための新しいインターフェース：NIME 研究は学術研究としてどのように位置づけられるかについての論考の中で、いくつかの指摘と問題提起を行う (**Dahl2015**)。まず、新しい楽器を作ることは意地悪な問題の一類型であるということだ。これは環境問題のような大規模な問題の類と比べれば全く同じような類の意地悪さではないようにも感じるが、最低でも明確な議論のガイドラインが存在しないということや、一般化できる事例が少な過ぎて理論化しづらいという意味では、明確に科学的事実を解明する研究パラダイムとは異なるという意味では納得がいく。

加えて、Dahl は Stokes の研究という行為を基礎的理解 (≒知の創出) のための研究であるか？という軸、実用されることを考慮されているか？という軸の 2 つで分割した四象限 (**Stokes1997**) を参照し NIME 研究の位置付けを試みる。図 3.1 において、知の創出かつ実用を目的としない左上の領域は例えば量子力学における Bohr のような基礎研究の領域であり、一方実用されることだけを目的とし知の創出そのものは目的としない右下のエジソンの発明群、そして、右上に実用的な理由に牽引されつつも新しい知の基礎となった Pasteur の乳酸発酵の研究などがある。

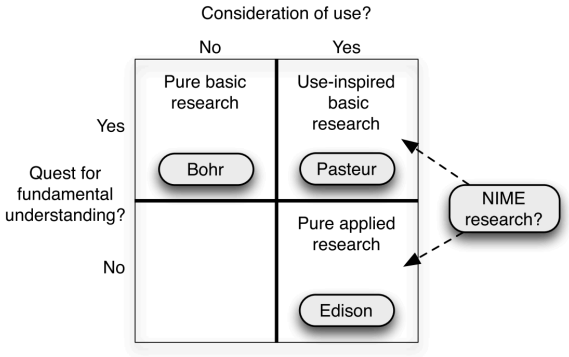


図 3.1: Stokes による研究の四象限。(**Dahl2015**) より。

この四象限に NIME 研究を位置付けるとすればどこになるだろうか？右下の実用だけを意識したものだとすれば、それは企業が生産する楽器群とどのように区別ができるのだろうかという問題がある。一方右上の実用を通じて知の創出を得る研究なのだとすれば、1. その知とは一体なんなのか、2. その知をいかに証明できるか、3. その知は一体いつ発生するのかといった問題群が浮かび上がってくる。

仮に NIME で創出できる知も Gaver がいうような Annotated Portfolio のように、理論として一般化することができず、個々の差異との比較によって議論されるものだとしたらいったん 1.(What) と 2.(How) は横に置いておいてもよいだろう。しかし 3.(When) は筆者のようなソフトウェアを研究するものにとっては考えるべき重要な問題になる。つまり、NIME 研究をするということは、楽器を作ることが研究なのか、その楽器を演奏する、あるいは評価実験などでユーザーに使ってもらうことが研究なのかということが混在していることを指摘している。

最終的に Dahl は Fallman のデザインリサーチの 3 つ組という分類 (**Fallman2003**) を利用して、企業が一般的に楽器を作るプロセスを” 産業とのインターフェース” である Design Process、楽器のデザインプロセスそのものについて研究する場合は” アカデミアとのインターフェース” である Design Studies、そして新しい音楽表現の探究のための一例としての NIME の提案という場合には” 広義の社会とのインターフェース” である Design Exploration（これは、Gaver の RtD の考え方と対応するものだと言えるだろう）という形に位置付ける。

音楽プログラミング言語全般の開発を研究とする場合にも、このような Design Studies としての性質と、Design Exploration 両方の性質を備えていると言えるだろう。Nishino は音楽プログラミング言語設計における知への貢献という側面が取り上げられていないということを指摘し、自らが開発する LC 言語における、microsound(Roads によって提案された、音声データを数ミリ秒単位に切り刻み再構成するような音声合成の手法 (**Roads2003**)) が既存の言語では十分に表現できないという具体的事象から、mostly-strongly-timed language という新しい音楽プログラミング言語における概念の創出ができているという具体例を示す (Nishino 2012)。これは Stokes の四象限におけるパスツールに相当するものと言える。もっとも、mostly-strongly-timed というコンセプト自体を引き継いだ言語は LC 以後開発されているわけではなく、果たして創出された知や概念というものがパスツールの発見した乳酸菌と同じように扱えるのだろうか、デザインを通じて作られた、究極的に個別的な人工物と区別がつくのだろうかという疑問は残る。

3.4 プログラミング言語における量と質の相互作用

こうしたデザインの個別性に伴う理論の一般化や評価の困難さは本論文で検討するもう 1 つの領域、プログラミング言語一般に関する評価について検討することでもう少し明確な議論をすることができる。

音楽目的に限らない、システムレベルのソフトウェア開発などに使われるプログラミング言語の設計を考えれば、その言語によって作られるアプリケーションの実行速度は明らかに重要な評価指標であり、また（使われるマシンの種類などの環境を揃えれば）定量的に計測することが可能である。では、あるプログラミング言語のコンパイラの機能 X の実装を工夫することにより、実行速度を 20% ほど改善できたとしたら、それはなんらかの事実を解明したことになるのだろうか、それともある手法やソフトウェアといった人工物を生成したことになるのだろうか？といったように、プログラミング言語という分野自体、1960 年頃から工学的なアプローチで長く研究がされている中、学術的研究としてどう正当化するか、特にどのように評価すればよいかという話題は 2009 年の ACM SIGPLAN（Special Interest Group of Programming Language: プログラミング言語の国際会議を主催する団体で最も大きなもの）主催のPLATEAU: Evaluation and Usability of Programming Languages and Tools が開催されるまであまり触れられてこなかった。

その中で Markstrum は新しい言語の実装や設計などのアイデアを論文として提示する際の正当化の方法として、**主張と根拠の整合性 (Claim-Evidence-Correspondense)** という見方を提示する (**Markstrum2010**)。Markstrum によれば、プログラミング言語の論文で提示される主張は大きく分けて 3 種類あるという。1 つ目はこれまで存在しなかった新機能を作ったというもの、2 つ目はすでにある既存の機能の効率性を上げるような内容、3 つ目は望ましい言語の特徴 (property)、つまりこの言語は直感的である、読みやすい、効率的であるといったような内容だ（代表的な表現として、プログラミング言語のシンタックスを形式的に定義する Backus-Naur 記法で知られる Backus が “Elegant” という表現を用いている (**Backus1979**) ことを挙げている）。しかしこれまでのプログラミング言語の論文には、1 と 2、つまり新機能と機能の増強に関しては論文が査読されたものであるならば十分認められるものであるが、望ましい言語の特徴に関してはその主張と、それを支える根拠は両方とも提示されているが論理的な結びつきが不十分であるとしたのだ。

こうした研究はのちにより近年の Coblenz らによるプログラミング言語における評価語彙の整理（第 5 章で詳しく紹介する）などにつながっており、Markstrum 自身も警鐘したように、主張と根拠の整合性を証明する方法論は統計的な評価実験のような実証主義的方法よりも、例えば近年の Muller と Ringler の修辞的フレームワークのように、論文中で主張された表現の変遷などを人文学的手法で辿ることで明らかにするような研究が進んでいる (**Muller2020**)。

つまりプログラミング言語という領域における知の創出の体系は、新しい文法の提示のような生成的な人工物、ベンチマークテストのような量的に計測、比較ができる結果、ユーザーインタビュー調査のような質的調査の結果、主観評価実験のような統計的手法による量的なユーザーフィードバックといった様々な種類の論拠を組み合わせて、効率が良い、読みやすい、表現力が高いと言った主張をなんらかの修辞的な方法によって結びつけることによって行われる。

このようにプログラミング言語という領域が、論拠に関してはある程度反証可能性が存在する、しかし最終的に作られた言語そのものの定量、定質的評価が難しいという形而上学とも形而下学 (≒科学) とも言える独特の体系を持つ理由としては、量的な特徴の変化がその言語の質的な変化に大きく影響を与えることが時にあるという事情もある。

例えば音楽プログラミング言語に関していえば、本章冒頭で” 合わせ技” と表現したように、アルゴリズムを用いた作曲を同じ手法でも、人間が手動では行えないような計算速度を借りることによって異なる種類の音楽にすることができるし、第 4 章で見るように、音楽プログラミング初期の一度信号処理の結果をテープに書き込んでから再生する方式と、80 年代以降のリアルタイムに信号処理ができる環境とでは作られる音楽の質も大きく異なる。加えて、より事情を複雑にする問題としては、例えばライブコーディングのような、プログラムの動的変更のしやすさという質的特徴を強化した言語を作ろうと思うと、プログラムの実行性能は静的な言語と比べると低くならざるを得ないという、いわば**量的特徴と質的特徴間のトレードオフ**が発生するという問題がある。

こうした事情を鑑みれば、ベンチマークや主観評価実験、インタビュー調査といったあらゆる手法による証拠そのものにはそれが存在するだけである程度一次資料として有用性もあるし、ただ実行速度が上がるような改善をもたらすだけでもその言語の存在意義に関わる変化となる可能性はある。そこから先の主張に関しては Gaver の言うような人工物の Annotation を補強するものとしてこれらの論拠を使うことで、個別の言語の具体性をより高め、理論化は個々の差異を見ることによって可能になる反証できない生成的学問として取り扱う、というような棲み分けをすることで RtD としてのプログラミング言語研究という領域を記述することができる。

この考え方は音楽プログラミング言語という、汎用プログラミング言語よりは量的に測れない指標が多くなってくる研究対象に関して、工学的アプローチ以外の研究方法の見通しを良くしてくれるものになるだろう。

3.5　音楽プログラミング言語のデザインという研究領域

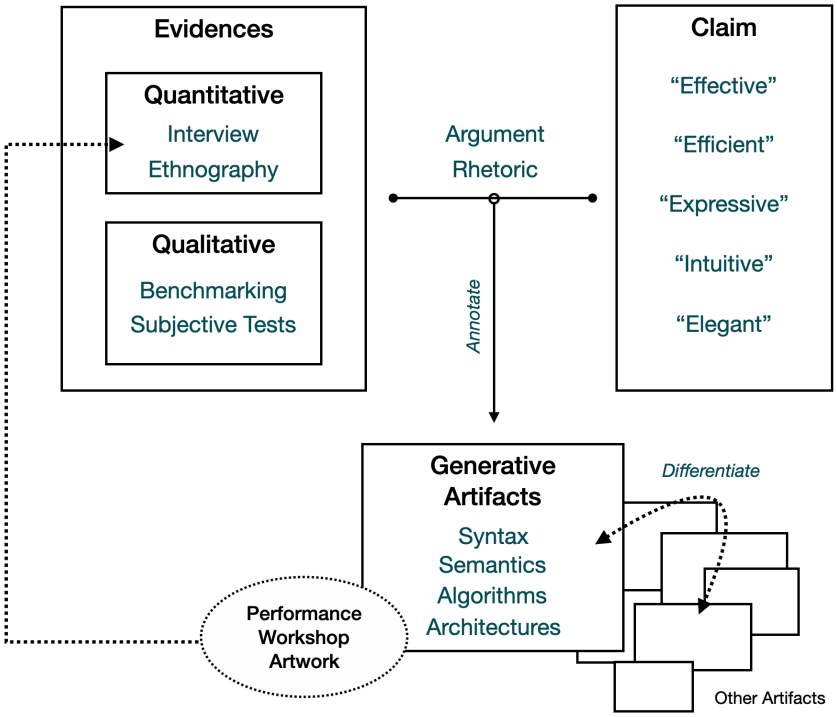


図 3.2: Gaver、Markstrum、Muller らの議論を統合した、RtD としてのプログラミング言語の研究の概念を表した図。

さて、ここまでの議論を図 3.2 にまとめた。プログラミング言語のための新しい文法やアルゴリズムを提案するという作業は時に既存の研究から飛躍することもある生成的な作業である。そしてそれによって成される主張は時に主観的な用語を用いる不明瞭なものでもある。そうした主張を支えるものとして、ある程度反証可能な方法で論拠を提示する。これには主にベンチマークや主観評価実験に基づく定量的データによる論拠、文献調査、インタビューとその分析、エスノグラフィと言った様々な手法によって得られる質的なデータによる論拠、時にその両方が存在することになる。その論拠と主張は適切な修辞によって結び付けられ、その claim-evidence の結び目を持って生成した言語やアルゴリズムという人工物に注釈：Annotation を付ける。これらの要素はある程度独立性があるため、例えば論拠となっているベンチマークの測定に間違いがあったり、質的データ分析の方法論が適切でなかったとしてそれらのデータを取り下げたとしても、作られた人工物そのものを丸ごと棄却しなければならないわけではない。新たにデータを分析し直して適切に主張と結びつけ直す

ことができればその言語の学術的価値を改めて提示することができる。

注意しなくてはならないのは、論拠はあくまでも主張と結びつくことによってはじめて意味があるものとなるので、単に言語 A におけるベンチマークと言語 B のベンチマークを比較して A のほうが性能が高かったからといって、言語 B は淘汰されるべきということにはならない。これは量的特徴と質的特徴のトレードオフの問題を考えれば当然とも言えることだろう。加えて、論拠には様々な研究パラダイムが混在しており、質的研究の中でも実証主義的—解釈主義的なスペクトラムが存在しているため、自分がどの立ち位置で論拠を見出そうしているのかを依然自覚する必要がある (**Otani2019**)。それゆえ、ベンチマークを取るのであればその手法や環境は再現可能性が担保されているべきだし、評価実験を行なったのであれば、その実験自体も追試が可能なものであり、実験によって得られたデータの分析方法も検証可能なものでなければならない。

加えて、音楽プログラミング言語という領域で考えれば、Dahl の NIME 研究における When の問題を考えれば、作られた音楽プログラミング言語を用いて作った音楽、プログラム、パフォーマンスといった二次人工物とでも言うべきものが、それ自体デザインの成果としての比較対象になる独立性を持ちながらも、その経験を通したインタビューやエスノグラフィといった、新たに一次人工物に注釈を付けるための材料にもなっているという、再帰的な構造を持っていることがわかる。

3.6　拡大するデザインの領域

ここまでは比較的 HCI という閉じた領域における RtD の実践からみた音楽プログラミング言語研究の位置付けと方法論について検討してきた。しかし近年、RtD の対象とする領域はこれまでのプロダクト、サービスといった単位のものから社会システム、法、金融などより広がりつつある。Koskinen はこの広がりつつあるデザインをその活動の場という視点で Lab、Field、Showroom という 3 つに分類した (**Koskinen2015**)。Lab における RtD は本論文でも中心的に議論の対象となっている、ソフトウェア、インターフェース、特に NIME 研究にも代表されるもので、生態学、認知心理学などに由来し、プロトタイピングと評価実験を繰り返しその価値を提示する。Field での RtD は文化人類学などに由来する、ユーザーがいる環境を社会科学的な質的な調査法を中心に理解することを目指した研究である。そして Showroom での RtD はいわゆるクリティカル・デザイン (**Marpus2019**) やスペキュラティブ・デザイン (**Dan2013**) に代表される、ありえるかもしれない並行世界に存在しうるような装置や、時にユーモアを交えた不合理さを持つ、必ずしも現代において役立つものではないプロダクトのような人工物を、物語、展示や映像といった表象を組み合わせることで、社会構造に対する批評的役割を担ったり、議論を促すことを目的としたアプローチだ。スペキュラティブ・デザインやクリティカル・デザインに通底する考え方は、現在の科学技術が中立的ではないあるイデオロギーのもとに出来上がったものだとして、Papanek の「生きのびるためのデザイン」(**Papanek1971**) に代表される、大量消費社会における功利主義的なプロダクトデザイナーに対する批判を受け継いだものである。

このようなデザイン領域の広がりと並列して、音楽プログラミング言語のデザインも単にコンピューターを使った音楽制作のための道具というだけの側面ではなく、コンピューターを用いる音楽文化そのもののグランドデザインといった側面を持つ研究が行われてきた。Mclean、Magnusson、Sorensen らのような、ライブコーディングと呼ばれるコードを書くことで演奏する文化を形成してきた音楽プログラミング言語の設計者たちはそうした音楽文化自体の（再）デザインを試みてきた人物である。本章の残りは、このような音楽プログラミング言語設計自体の再構築する試みを広義の RtD 的実践として位置付け、さらにその背後にある科学技術社会論の理論と接続することを目指す。なお本章で取り上げる TidalCycles、IXI、Extempore という言語はその実装面を技術的にも様々な議論の切り口はあるものの、その視点は第 4、5 章へ先送りし、本章ではその言語の目指す音楽文化のビジョンという大きな射程の方により重きを置く。

3.7 科学技術社会論との繋がり

クリティカル・デザインの背景には科学技術社会論（STS）と多くの理論的系譜を共有する (**Marpus2019**) とされており、中でも 80 年代にブルアが提唱したストロングプログラムに代表される、社会学の理論を科学にまで拡張させ、科学知識の理論もある社会集団によって形成された信念の一種であるため、その集団の特性やプロセスの分析によって科学や技術の実態を明らかにすることができるという社会構築主義的な態度と並列して発展してきたものと並行して発展してきている。一歩進めて、分析からデザインという人工物を作ることにより、開かれた環境で議論を巻き起こし未来の科学技術環境をかたどる公衆を形成するものとして

3.8 Alex Mclean(TidalCycles)

3.9 Magnusson

SuperCollider を用いたライブコーディング環境 ixiQuarks を制作した Magnusson は博士論文でコンピューターを用いたデジタル楽器（DMI:Digital Musical Instruments）を特徴付けるものはなにか、つまり、アコースティックな楽器と DMI はどう違うのだろうかというシンプルな問いを、Stigrer の技術哲学、Latour、Callon らによるアクターネットワーク理論などを援用しながら DMI の現象学的考察を行った (Magnusson 2009)。これはコンピューター音楽のためのプログラミング言語や DMI 研究の中でも特異かつ長大な人文学的考察を行っている。その中で彼は**認識論的道具 (Epistemic Tool)** という概念を提唱した。この認識論的道具とは、別の表現では**拡張された精神によって使用される、知識のコンベヤー**ともされており、なるべく単純に言い直すのであれば、コンピューターという、シンボルを操作する装置を使って楽器を作る以上、その楽器の中にはその文化圏でやりとりされるシンボルが埋め込まれることになるという現象を表した言葉である。

Magnusson が射程に置いたものの 1 つは、DMI を演奏するときの身体の役割はどこにあるのかという視点、それから、演奏者における主体はどこにあるのか、ということだと解釈できよう。前者の、DMI における身体性というのは NIME 研究の中でも重要な話題であり、90 年代にラップトップによる個人での音楽制作が容易になって以降、演奏においてラップトップに向かいトラックパッドやつまみの付いたコントローラを操作するだけのような傾向に対して、より人間の身体を十全に活用できるようなインターフェース作りが目指されてきたという背景がある (Jensenius and Lyons 2016, p. 4.2)。

確かに、コンピューターを用いた楽器は身体の様々な動きをセンサーなどを通じて取り込み、様々な音楽的要素にマッピングすることでアコースティック楽器とは異なる新しい表現を可能にするかもしれない。Tanaka は DMI におけるパラメーターのマッピングこれを認知心理学者 Gibson が提唱し Norman がインターフェースデザインの分野に援用したアフォーダンス理論、つまり主体が行動できる可能性が環境やオブジェクトの中に埋め込まれているという考え方 [ˆaffordance] を用いて分析することで、DMI を演奏するという時には、楽器が演奏者の音楽的表現を引き出すというアフォーダンスと同時に、それを見る/聴く聴衆との間に音楽を通じた対話的なやりとりを引き起こすアフォーダンスという、2 段階のアフォーダンスが存在していると主張する (Tanaka 2010)。

この主張は、楽器のデザインにおける視点を演奏者–楽器という閉じた系から、演奏者–楽器–聴衆という系として、DMI 制作を社会的行動の 1 つとして視点を広げるものであり、実際 Magnusson 自身も Tanaka の論文で引用されるように ixi を用いたパフォーマンスをアフォーダンスの観点から分析している (**Magnusson2006**)。

Magnusson の認識論的道具はこの社会的行動としての道具をさらに、演奏者–楽器–聴衆の三者の一回きりの関係とするのではなく、プログラマが長い時間をかけて楽器を作る作業の中に、その中で他の人がさらに時間を掛けて開発してきたライブラリを用いて開発し、繰り返しパフォーマンスをする中で演奏者と聴衆の間で共通の語彙が形作られていく、という、長い時間をかけて培われる音楽文化の中の行動のひとつとして DMI、

あるいは ixi という音楽プログラミング言語制作を位置付けたわけである。だがここ Magnusson が指摘するのは DMI を作ることが社会的行為である（特に、センシングなどを用いた参加型の楽器やパフォーマンスなど）ことを肯定するよりも、むしろ DMI の制作者はアコースティック楽器に不可能な自由な音楽制作を一見しているように見えて、その実記号的循環の中で形成される社会の文脈に拘束されているといった否定的ニュアンスの方を強調している。

> アコースティック楽器制作者とは反対に、構築/作曲された（composed）デジタル楽器のデザイナーは、シンボル的デザインを通じアフォーダンスをかたどるため、それゆえ音楽理論のスナップショットを作り音楽文化を時間的に凍結させるのだ。(Magnusson 2009)

この悲観的とも言える態度は科学技術社会論の潮流の変化と照らし合わせることで理解しやすくなる。社会構築主義的な見方によって科学技術はある社会集団のイデオロギーによって形成されるのだとすれば、例えば楽器の材料のような物質的な、より端的に言えば自然の中にあるものは作るものに何も影響を与えないのだろうか？例えばこうしたアプローチの代表とも言える Pinch と Bijker の技術の社会構成論 (Social Construction of Technology:SCOT) では、Magnusson が念頭におく Latour のアクターネットワーク理論のような、非人間が人間の区別を取り払い（アクターとすることで）、それらが相互に影響を与え合うという視点には、そのような社会一元論的な考え方に対して、自然–社会とそれぞれが与え合う影響のバランスを取る意味合いがある (**Fukushima2017**)。

この価値観の中では、楽器は楽器製作者の意図それだけによって構築されるのではない。アコースティック楽器の制作者は木材や金属といった材料を相手にし、その音色や演奏される方法は制作者と素材というエージェント（無生物）との折衝において形成されていく。また Magnusson は電気楽器であってもやはり電子部品という素材と格闘しながら反復的試作プロセスを繰り返す中でデザインが固まっていく。ここでは、例えばオシレーターやフィルターと言った、科学技術の用語とメンタルモデルを組み合わせることで楽器は構築されていくため、素材との対話といった側面はやや落ち着くものの、物理的制約や不確定性は未だ残るとされている。

では果たしてコンピューターを用いた楽器の場合はどうだろうか？コンピューターという象徴機械におけるプログラミング作業はもはや不確定な物理的特徴による制約がない代わりに、材料となりうるものは言語や記号そのものであり、しかもそれはほとんどが誰かが作り上げてきた言語体系をブラックボックス化したもの（≒ライブラリ）だ。結局、“想像しうるどんな音でも作れる”、というコンピューターを用いた音楽の売り文句であった言葉は、裏を返すと “想像することができなければどんな音も作ることはできない” という対偶によってその可能性に限界をはじめから示しており、むしろ材料や物理的特性による意図しない変化や使用法を生み出すことができないという意味では、さらに広がりがないものにもなりかねないのだ。

3.10　隠されたアフォーダンス、知覚されるアフォーダンス

このアコースティック楽器とデジタル楽器の特徴の違いはアフォーダンスが「真のアフォーダンス」か「知覚されたアフォーダンス」かという視点でも同じことが説明できる。

Gibson が認知心理学の分野において導入したアフォーダンスという概念と、それをデザインの分野に援用した Norman のアフォーダンスの概念はかなり異なることが知られており（Tanaka の文献でも明確に区別されている）、Norman もこれを後に自分でも別物だと認識し、Gibson の提唱した本来のものを「真のアフォーダンス」、自らがデザインに応用したものを「知覚されたアフォーダンス」と言い直し (**Norman2000**)、さらに後年では明確にアフォーダンスという語を導入したことが失敗だったと話し、知覚されたアフォーダンスという名前も「シグニファイア」と新たに名付け直している (**Norman2011**)。

Norman の解説を借りれば、Gibson の意味するアフォーダンスとは “生態の持つ可能性とモノの持つ

可能性との間の関係性であって、それらは存在に気づくか気づかないかには関係なく実世界に存在するもの”(**Norman2011**) であり、一方の知覚されたアフォーダンス（シグニファイア）とは、例えば Web ブラウザの縦スクロールバーは上下方向にしか動かないことでこのページが左右にはスクロールできないことを示し、ユーザーの行動を補助していると見ることができ、これはプログラマが付けた制約によって生まれた知覚がユーザーの行動をアフォードしているといったようなものだ。“知覚されたアフォーダンスは、現実のものよりも約束事に関係することが多い”(**Norman2000**) という説明と、後の言い換えのシグニファイア(Signifier)とは記号論におけるシニフィアン(Signifian)そのものであるように、信号(Signal)を伝える記号的な、あるいは記号によらない形式的な仕組み—例えばギャロウェイの言うプロトコル (**Galloway2017**) などを用いて伝達する仕組みを表したものだと言えるだろう。

アコースティック楽器を制作する際に楽器製作者は作りたいものを想像して作っているかもしれないが、現実の制作は木材や金属、あるいは電子部品といった材料にアフォードされてその材料特有の性質が決定される。

一方コンピューターを用いた楽器をプログラムを組むことで作る際には、誰かが作ったライブラリの API–Application Program **Interface** に持たせられたシグニファイアに従って

第 4 章

What1: 音楽プログラミング言語の歴史

4.1 音楽プログラミング言語の歴史

ここまで、第 2 章では、コンピューターがメディア装置として扱われるようになるまでの歴史と、その中で培われてきた、コンピューターは不可視になってゆくべきだという思想についてを紹介した。そして第 3 章では音楽の流通インフラストラクチャの変遷の歴史を辿ってきたのだった。

本章では、この 2 つの歴史を踏まえた上で音楽プログラミング言語とはどのような特性（≒ナラデハ特徴 (by 松永)）を持っているのかについてを再整理する。

まずは、既存の音楽プログラミング言語に関するサーベイの内容について軽く触れておく。音楽プログラミング言語に関する文献は基本的に、（本論文もそうだが）個別のプログラミング言語実装についてが主内容の論文の前段として書かれているものがほとんどだが、サーベイに特化した文献としては (Nishino and Nakatsu 2015) と (Dannenberg 2018) が存在する[*1]。

Nishino と Nakatsu のサーベイでは 1940 年代の電子計算機誕生直後から試みられてきた、コンピューターを音楽に用いるための歴史を時系列に追いかけている。Dannenberg のサーベイでは、同様に歴史的変遷を辿った上で、各言語を特徴づける要素を Syntax、Semantics、Library、Development Environment、Community &Resources という 6 つの要素として提示し、また、汎用プログラミングでは考慮されることの少ない音楽特有の課題を列挙し、代表的な言語間での記述の違いを分析している。

また (田中 2017) は 70〜80 年代におけるチップチューンと呼ばれる、初期のパーソナルコンピューターやゲーム機においてとられた、音声生成用の IC チップを用いた音楽についての歴史を主に解説しているが、40〜60 年代までのコンピューター音楽黎明期についての記述も厚い。

本章でも先行文献と同様に歴史的に代表的な言語を時系列に紹介するが、既出の文献では言及されていなかった点として、以下の 2 つの項目に着目し、大きく時代を 1970 年、1990 年ごろを大きな区切りとし、3 つに区分けして整理する。

1970 年代の区切りは 2 章で見てきたコンピューターをメタメディア装置として扱う思想の始まりと、パーソナルコンピューターの登場という 2 種類の出来事である。

そもそも音楽プログラミング言語の祖先となるソフトウェア MUSIC が開発されたのは正解で最初の (汎用) プログラミング言語 FORTRAN が作られた翌年であり、当然 1 章で見たアラン・ケイらによる対話的プログラミング環境や豊富な入出力インターフェースを備えるよりもずっと前のことである。なので、必然的に音楽ソフトウェアのプログラミング自体も機械語を直接入力するかアセンブリ言語 (機械語の命令をテキストと 1 対 1 対応させたプリミティブなプログラミング言語のようなもの) しかなかったし、そのソフトウェアに対する入力データ（≒楽譜）も同様の形式を取らざるを得ないものだった。

[*1] なお、Nishino は LC(Nishino, Osaka, and Nakatsu 2014) の、Dannenberg は Nyquist(Dannenberg 1997) 他多数の言語の設計者でもある。このことからもやはり、音楽プログラミング言語の歴史を記述するには基本的にその設計や実装に関する知識や経験が必要になっていることが窺える。

つまり、1950〜1970 年代の音楽プログラミング環境は大まかにいってコンピューターで音楽を作るためのソフトウェア全般の祖先にあたるものであって、必ずしもプログラミングという行為やテキスト入力という形式の固有性を積極的に取り入れたものではない、ということだ。

逆に、70 年代以降の音楽プログラミング言語/環境はマウスや (文字入力や、ピアノ鍵盤どちらにせよ) キーボード入力といった直感的（WYSIWYG 的）なインターフェースが選択肢として存在する中で敢えてプログラミングという手段を使うものとして設計されてきた、という違いがあると言えるだろう。プログラミング環境であっても、GUI の誕生は Max(Puckette) を代表としてテキストインターフェースだけでなく、入出力を持つボックスをマウスで繋いでいくような形式など、テキストに留まらない形式でのプログラミング行為を可能にした。これは同時に、出力された信号などもオシロスコープのようなグラフィックとしてフィードバックが返ってきたり、パッチ（Max におけるプログラムのこと）中にスライダーのような、プログラムされたソフトウェアを操作するためのインターフェースが同居していたりといった、それまで存在していた” プログラムを構築するステップ” と” 構築されたプログラムを使用するステップ” に明確な境目が無くなっていく歴史でもある。

また、1990 年代の区切りは、パーソナルコンピューターが専用のサウンドチップなしに、CPU だけで音声信号処理をリアルタイムで行えるようになったこと、そして汎用プログラミング言語の理論が音楽向けの言語にも流入し始めたことの 2 種類である。

チップチューンを歴史に入れること

コンピューターアーキテクチャのスタンダード（もっと言ってしまえば、x86 アーキテクチャ）が定まるまでのプログラミングは、特定のハードウェアのための特定のプログラムを作るという側面が大きく、書かれたソフトウェアが様々なプラットフォームで使い回しが効くということでもなかったことも頭に入れておくべきだろう。汎用プログラミング言語はそれまでの実在するハードウェアに対する命令列を可読性のあるテキストデータから出力するためのソフトウェアという側面だけでなく、ラムダ計算（引用）のような、計算過程自体を数学的になるべく普遍的になるように記述する代数学の理論との接続を見せるようになり、LISP や ML、Haskell に代表されるような関数型プログラミング言語のパラダイムが発生してきた。そしてこうした分野で培われたプログラミング言語の理論は現在では Faust や Kronos を代表とする、関数型でかつ音楽や音声処理のための言語の理論的基盤としても用いられるようになっている。

つまり 2020 年代現在において、本論文が定義する音楽プログラミング言語とは、**コンピューターを用いて音楽を生成するためのソフトウェア群に始まりつつも、並行して発展してきた汎用プログラミング言語やその理論を取り込みつつ発展してきたソフトウェアやツール**のことを指す。なので、Max のように前者の流れを強く汲むものは、Dannenberg が言うように、言語体系とランタイムやライブラリ、開発/実行環境があらかじめ切り離せない形式 (＝実装そのものが仕様) となっていることが多い。逆に、汎用プログラミング言語の理論をベースに構築された言語、例えば Extempore や Faust、Kronos では、言語仕様は言語仕様として独立しておりランタイムが存在しないーあるいは複数のランタイムの実装があり得る、そのほか、決まった IDE が存在しなかったり、複数の開発/実行環境が存在するといった構成になっているものがある。

図を入れる

なお、本稿では既存のサーベイでは特に 70 年代以前の研究所レベルのコンピューターを用いた取り組みに関しては Nishino らのサーベイに十分詳しい記述がなされているので、本稿に大きく関係しないと事例については省くことにし、より記述の少ない 2000 年、2010 年代に作られた言語について積極的に取り上げる。

また、用語として” 音楽プログラミング環境 (Music Programming Environment)” と” 音楽プログラミング言語 (Music Programming Language)” という 2 つの言葉は特に明示的な使い分けをされずに使用されることも多い（英語では、Computer Music Language や Computer Music Environment、Computer Music Systems などなど）が、本稿では、“音楽プログラミング環境” といった時には開発環境やライブラリなど、その言語を用いる時に利用するツール全体に重点を置きたいときに、” 音楽プログラミング言語” といった場合

にはその言語仕様や文法など、ソフトウェアというよりも言語そのものに着目したいときに、用いることを附しておく。

4.2　研究所レベルでの取り組み

世界で初めて音楽にコンピューターを利用した例、というのを考えるのは、世界初のコンピューターは何かという論争や、世界で最初のアルゴリズミック・コンポジションとは何かといった命題につながってしまいキリがなくなってしまうので、ひとまず本稿で扱うのは ENIAC 以降の電子計算機、つまり真空管を用いて HIGH/LOW の 2 値をスイッチングすることで任意の計算を行える機械が誕生して以降の計算機群についての事例に限定することにする。

はじめてコンピューターを用いて音楽を鳴らした最初期の例としては、イギリスのコンピューター BINAC、アメリカの UNIVAC I、オーストラリアの CSIRAC などが挙げられる。これらはもっぱらデバッグ目的で取り付けられていたスピーカーに 2 値の信号をマスタークロックの周波数から逆算して規則的な周期で送ってあげれば、任意の音程の信号が出せるだろうという考えで作られたものだ。

そのため、任意の音程やリズムを奏でることはできるが、レコードのように任意の音圧波形を想いのままに作れるというわけではなかった。

コンピューターを用いて任意の波形を生成するという課題に最初に真正面から取り組んだのが Bell 研究所の Mathews らによる MUSIC シリーズだった。

MUSIC がそれ以前のシステムと異なっていたのは、パルス符合変調（PCM）と呼ばれる、音声波形を一定時間に分割（標本化）、各時間の音圧を離散的な数値として表す（量子化）、今日のコンピューター上における音声表現の基礎的な方法に基づいた計算を行ったことだ。パルス符合変調の元となる標本化定理はナイキストによって 1928 年に示され (Nyquist 1928)、パルス符合変調は Reeves により 1938 年に開発されている。

この考え方を簡単に表したのが Hartley1928 における図 n である。時間を横軸、音圧を縦軸にとった音声波形のグラフをグリッド状に区切り、連続した数値を離散化された数値のリストに変換する。区切るグリッドが少ないほど、実際の波形との誤差が量子化歪みとして現れる一方、グリッドを細かくするほどに必要なデータの量は増えていく。また、標本化定理より、例えば 1000Hz の周波数成分までを持つ波形を標本化するとき、その 2 倍である 2000Hz 以上、つまり横軸のグリッドを 2000 分の 1 秒より細かく設定する必要があることが知られている。例えばサンプリング周波数が 1800Hz だった場合、表現できるのは 900Hz までとなり、1000Hz の正弦波をこのサンプリング周波数で標本化すると $1800 - 1000 = 800Hz$ の信号が折り返し歪み (エイリアス信号) として現れてしまう。

人間の知覚できる周波数の上限が 20000Hz 程度となっているので、その 2 倍である 40000Hz 以上の標本化周波数で、かつ量子化歪みが十分に少なくなるように量子化ビット数を決めておけば、人間が近くできうる範囲ではおおよそどのような波形でも数値として表現できるということになる[*2]。

実際には連続した波形を離散化するだけであればこのような考え方の考慮で十分なのだが、標本化/量子化した波形同士を演算する場合にはもう少し細かい事情を考慮する必要があるため問題点も指摘されてきており (Puckette 2015)、音楽プログラミング言語設計の根幹にも関わってくるのだが、それはのちに議論する。

[*2] 例えばコンパクトディスクの規格などはこういった考慮から標本化周波数 44000Hz、量子化ビット数 16bit と定められている。

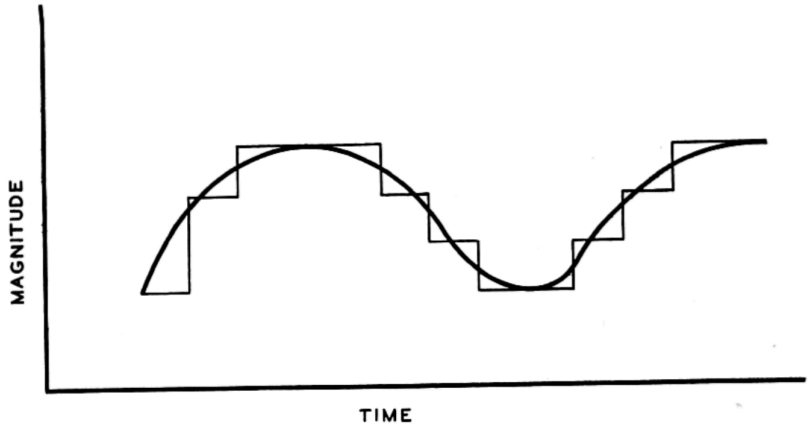


図 4.1: Hartley の論文における PCM の概念を表した図。

MUSIC は I から V までの 5 バージョンが存在している。

1957 年に作られた MUSIC I は IBM 704 というコンピューター上で動作する、対称系の三角波の波形に対してエンベロープを掛ける程度の波形の合成ができるシステムだった。この頃のシステムはリアルタイムで波形を電気信号として生成しスピーカーを鳴らせるようなものではない。まず MUSIC のプログラムをコンピュータが利用できる IBM 本社まで持ち込み、計算結果を磁気テープにバイナリデータとして書き出し、磁気テープをベル研究所に持ち帰り、そこにあった真空管製の 12bit デジタル-アナログコンバーターに通してはじめて音が出せるようなシステムになっていた (Max Mathews and Roads 1980; Roads 2001)。

MUSIC I の時点でのサンプリング周波数について記述されている文献はいないが、後年の IBM 7090 上で動作していた MUSIC IV(1963 年) では、同様に 12bit での量子化で、実際の計算速度が 1 秒間に 5000 サンプル程度が性能の限界だったのに対し、実際に計算結果を再生するときにはそれを 1 秒に 30000 サンプルに早回しすることができたと記述が残っている (M.V. Mathews 1963)。

Mathews は MUSIC を制作するにあたって、シャノンらが示した標本化と量子化によって、考えうるあらゆる種類の音が計算によって生み出せる、それもこれまで既存の楽器では奏でられなかったような新しい種類の音が作れることに強く関心を持っていたことを語っている。

> Essentially the sampling theorem shows that there are really no limits to the sounds you can make from samples. Any sound the human can hear, you can make with the right number, accuracy, and combination of samples, so the computer is a universal instrument. Other instruments, the violin in particular, are beautiful, lovable, but they always sound like a violin—or at least it's very difficult to make them sound not like a violin.(Hong Park and Hall 2009)

また彼は同時に、自身がバイオリン演奏をしていたが決して演奏が巧くはなかったため、身体的卓越を必要としない仕組みを作りたかったこと、さらに作曲家が曲を作ってもオーケストラに演奏してもらえる機会がなければ発表できないことに対するひとつの解として、作曲したものをコンピューターに演奏させるという 2 つのモチベーションも挙げている。

> My interests came from two things. One was that, although I've always loved to play the violin and I've had an amateur string quartet going most of my life, I was never very good at it, and so I wanted to be able to make better music that didn't require such manual dexterity as almost all [musical] instruments require. I also felt that there were many composers who would compose a piece for an orchestra and who would never hear the piece, or its performance would be delayed for years, and so [the computer] would provide a way for composers to write something and hear

it almost immediately.

これらコンピューターを音楽に用いる 3 つの理由を簡単に言い換えると次のようになるだろう。

まず 1 つはこれまで存在しなかった新しい音楽の探求だ。これはサンプリング定理に基づくものでない、音価レベルでのアルゴリズミック・コンポジションにも共通して言えるモチベーションと言える。2 つ目は身体拡張、あるいは自動演奏の追求である。これは今日の New Interfaces for Musical Expression における研究のような、コンピューターを用いて音を生成するための身体との境界面：インターフェースを作ることによって音楽と身体の新しい関係性を発見するものだ。これは最終的に 1 つ目の新しい音楽表現につながることもあるが、逆に、表現自体はこれまでも存在していたが演奏に高度な技能が必要とされるものを誰でも演奏できるように...といったモチベーションにも読み替えられる。そして最後の作曲家のためのオーケストラに代わるものとしてのコンピューターとしての視点は、クイック・プロトタイピングのしやすさと表現するのがわかりやすいだろう。

ただし、もう少し本論文の興味に引き付けて読んでみると、**コンピュータ以前はオーケストラという装置が作曲のインフラストラクチャとして機能していた**という視点を導入することもできるだろう。つまり、有名な作曲家であればオーケストラに頻繁に演奏してもらえる機会に恵まれ、そうでなければ作曲家は楽譜上に音を配置し、ピアノなど少ない数の楽器である程度のシミュレーションをしながら作曲することしかできないという状態だったと言える。特に実験的な作品、すなわち既に共通して使われている楽譜上の記述では足りない表現を指示しようとしたり、大人数が演奏してみてはじめて曲の修正ができるような複雑な作品を作ろうとしたときにはオーケストラに実際に演奏してもらえるか否かは思い通りに作品が作れるかどうかに大きく関わっていたということができるだろう。

MUSIC シリーズにおいて触れておくべきことは、MUSIC III においてはじめて **Unit Generator** と呼ばれる、今日まで用いられる概念が登場したことと、MUSIC IV においてはじめてそれ自体の実装が汎用プログラミング言語 (FORTRAN) で実装されたことである。

まず、Unit Generator とは〜

4.3　ボーンの IRCAM 4X プログラミングの分析

リアルタイムで処理できない程度に重かったこと楽器のモデルを計算できる理論が存在しなかったこと→Risset に始まる物理モデリング合成研究（Analysis-Synthesis アプローチ）

80 年代にしては計算のシンプルさに対して複雑な音色が出せる FM 合成とかはあったが、コンピューター音楽言語は大学や研究所に限られていたし、リアルタイム性にもまだ欠けていた extreme mediation, both temporal and conceptual

モデルがしっかりしていないと音を改善することができないけど、トライ&エラーに時間がかかるのでモデルの妥当性を確かめるのにも時間がかかってしまうパラドックス

学生が Cmusic で適当に音量をデカくしたせいで foldover 歪みが発生していたが、それが案外良かった→しかしシステム側で発生した歪みだったので再現できない本当はなんでもできるはずのコンピューターが何故！

技術のトリクルダウン　研究所でパイオニア的テクノロジーが発達し、それがコマーシャルに低価格化していくという考え方を IRCAM は持っていた

ヤマハの人が CX のデモに IRCAM にきた話

4x は当時最強のスペックだった→これ何がそうさせたんだろう？オシレーターとかはソフトウェアで仮想化できたんだろうか？西野の文献読む必要あり→できた。Variable Digital Signal Processor の話

OS から作ってた　ハードは凄かったがソフトとペリフェラルが弱い

Chant 歌声合成、Formes Patchwork と OpenMusic の手前

Chant ／ Formes のグループからは、音楽概念の高度な発達というコンピューターのポテンシャルを無視し

ていると思われてた

Chant/Formes は LISP 製、VAX/UNIX システムで動いていたノンリアルタイムシステム users could create their own “personalized environment” object oriented

The use and the development of software involve the writing of coded instructions within a software language or the creation of a completely new language, within the context of a hierarchy of such languages. At each of the hierarchy a traslation occurs between any two adjacent language or levels of code. Instructions from the language at a higher level mus be translated into aform whereby they can be “read” and executed by the lower-level code or language without any (or with minimal) loss of “meaning”

The hierarcy of codes that normally operates in computer software include, at the lowest level, machine code, the instructions that drive athe hardware, writtedn in binary form; at the next level up, assembler code, made of mnemonic abbreviations of machine code; above this, the general operating system that provides a basic framework and set of servicies; and above this, any of the major lkanguages such as FORTRAN, Pascal, C, or LISP.

〜〜

Computer music software such as that used and produced by IRCAM adds yet a further level of mediation, hierarchy, and translation, sincce the music languages are themselvels based upon, or written in, established general languages.

Thus, Music V is written in FORTRAN, Cmusic in C, IRCAM’s Chant in FORTRAN, and Formes in LISP.

Chant を使うには FORTRAN の知識も必要だったし、Formes を使うには LISP の知識も必要だったので、それを勉強するためにまず LISP について勉強しなくてはならない→**つまりこの時点ではまだ、後に McCartney が SuperCollider の設計指針として挙げる、プログラミングというコンピューターハードウェアを使うための専門的知識が必要な事項を encapsulate し、音楽の Notation に集中させるという意味での音楽” 言語” の概念は達成されていなかったということが言えるだろう。**

4.4　90 年代

音楽プログラミング言語の歴史において 90 年代は、パーソナルコンピューターでも高性能化と低価格化を背景として、4X のような研究所でしか使えない高級なハードウェアでなくとも、DSP 専用の計算ハードウェアを用いない CPU 処理でのリアルタイム音声合成が可能になった時代である。

ここでの高性能化と低価格化に関しては、具体的にどの製品の登場によってブレイクスルーが発生したという具体的な事例があるわけではないが、例えば久保田は「PowerPC G3 や Pentium II といった CPU によって、CD クオリティのサウンド処理に必要な、ある種のスレッショルドを超えたのだろう」と説明している [Kubota2017,88p]。

4.5 2000 年代

4.5.1 より詳細な時間制御（ChucK、LC、Gwion）

4.5.2 低レイヤの拡張（Faust、Extempore、Kronos、Vult、Soul）

4.5.3 高レイヤの拡張（TidalCycles、Sonic Pi、IXI、Gibber、Foxdot、Takt、Alda）

4.6 小括

音楽プログラミング言語は、元々は 50~60 年代の MUSIC N シリーズのような音楽をコンピューターで使うためのシステムとして生まれたものであるが、それはリアルタイムで音を生成できるわけではなく、磁気テープに書き出した結果を改めて再生するようなシステムだった。そしてこれは必ずしも今日の音楽プログラミング言語に限らず、DAW のような音楽制作ソフトウェア全般の祖先となるようなものだったと言えるだろう。

やがて 70~80 年代には、音楽プログラミングの歴史は 2 つに分岐する。一つはの時期普及し始めたパーソナルコンピューターにおける、音声合成 IC チップを用いたアマチュアを中心とするチップチューン、もう一つは IRCAM に代表される研究所レベルにおけるプロフェッショナルな現代音楽の文脈における研究、こチップチューンに使われた音声合成チップは非常に限られた数のオシレーターに対して CPU がその周波数や発音タイミングの命令を行う構成をとることで、計算コストのかかる音色の制御をハードウェアに任せて（既に成立している音楽体系である）五線譜的な表現に注力できるようにした。一方で IRCAM での取り組みでは、4A,4B,4C と言ったハードウェアではチップチューン同様にハードウェア的にオシレーターの上限が決まっているような構成から、4X でハードウェア的にはオシレーターを持たず抽象的な計算ユニットだけでリアルタイム音声合成ができるようになったという進展が見られた。チップチューンと IRCAM での取り組みを対比すると、この時代の取り組みはいずれもリアルタイム性を重視する代わりに、表現の自由度かハードウェア的コストのどちらかを犠牲にしていたと見ることができる。チップチューンではオシレーターの数を超える発音はできないし、4X は自由度が高い代わりに OS から独自で構成された非常に複雑な構成をしていた。このようにどちらかを犠牲にしてでもリアルタイム性を重視したのは、音楽をコンピューターを用いて生成するにあたって、人間は何かしらの記号、シンボルを解すことでしかコンピューターに対して命令を与えることができないという理由があった。つまり、音楽を生成するモデルを人間が考えコンピューターに実行してもらうプロセスが必要になるが、このモデルが妥当かどうかを検証するにはまたコンピューターに実際に実行してもらうプロセスが必要となる。（特に IRCAM のような新しい表現を追求する場においては）例え金銭的にコストがかかったとしてもリアルタイムにトライアンドエラーができる場所があることが重要だったと言えるだろう。

そして、90 年代には、パーソナルコンピューターでも高性能化と低価格化を背景として、DSP 専用の計算ハードウェアを用いない CPU 処理でのリアルタイム音声合成が可能になった。またこの時期にパーソナルコンピューターに用いられるオペレーティングシステムも Windows、Macintosh などに落ち着き、一つの CPU の中で複数のアプリケーションを実行するマルチタスキングのシステムもインフラストラクチャとして安定してきて、ユーザーアプリケーションとして音声合成を扱う場合にはプリエンプティブスケジューリングの中で可能な範囲でのリアルタイム性の追求という形に落ち着いたと考えられる。またこの時期に SuperCollider が汎用プログラミング言語において進んできた構造化プログラミングの手法を音楽プログラミングのために取り入れた。ここで初めて、音楽プログラミング言語はプログラミングという専門的なタスクから音楽、音声合成の抽象化という行為を引き剥がす機能を持つようになったと言えるだろう。

2000 年代以後の音楽プログラミング言語は大きく分けて ChucK や LC に代表されるプリエンプティブスケジューリング環境下での正確なイベント制御、Faust に代表される UGen レベルの低次の自己拡張性の追求、SuperCollider クライアントに代表される高次の自己拡張性の追求おいう 3 つの方向性があったとまとめ

られる。

第 5 章

What2: 音楽プログラミング言語における諸用語の整理

5.1 導入

第 3 章では時間軸に沿って音楽プログラミング言語の歴史を追うことでその特性を記述してきた。本章では、対照的に、現在利用されている音楽プログラミング言語の同士の比較をすることで、その共時体とも言えるものを検討する。つまり、音楽のためのプログラミング言語はその幅広い目的に応じて様々な形式の実装がなされている中で、目的や応用以外の視点で言語同士の比較のための概念や語彙を選び出すことによって、歴史的視点とは異なる形で「音楽プログラミング言語とはなんなのか」という疑問に答えるということだ。

本章の構成は大きく 2 つに分かれる。前半は音楽プログラミング言語を実装するパターンを、Spinellis らにおけるドメイン特化言語全般におけるデザインパターン分析を足がかりとしながら、実際の言語における構成を例を挙げながら検討することで類型化を試みる。また後半では音楽プログラミング言語における設計目標や評価に使われるGeneral、Efficient、Expressive と言った語彙が、同じ単語でありながらも異なる意味で用いられていることを指摘し、より適切な設計目標、評価のための語彙の形成を試みる。そのために、Coblenz らによる汎用プログラミング言語における設計目標の整理をもとに、音楽プログラミング言語を使用する過程を言語の実行環境が実際に動作する過程と組み合わせた、Human-in-the-Loop モデルとして提示し、各要素においてそれぞれどの設計目標が対応するかを検討する。

5.2 実装面から見た音楽プログラミング言語の実行環境

本章で検討するような言語の特徴の比較の先行例としては、Dannenberg のサーベイが最も詳しい (Dannenberg 2018)。

Dannenberg は音楽プログラミング言語を特徴づける要素として以下の 6 つの要素を挙げた。

- シンタックス（Syntax）
- セマンティクス（Semantics）
- ランタイム
- ライブラリ
- 開発環境
- コミュニティやドキュメントなどの資料

いわゆる「言語仕様」と言った時には、**シンタックス（Syntax）**と**セマンティクス（Semantics）**という 2 つを合わせたものを指すことが多い。シンタックスとは言語の表面的な仕様を表すもので、セマンティクス

とはよりプログラムの意味に直接関わる言語仕様のことを指す。

専門的な違いは文献具体例でこの違いを表そう。javascript という言語でで入力を 2 乗して返す関数 power を定義すると次のようになります。

```
function power(input){
  return input*input;
}
```

このソースコードにおいて function という単語は javaScript の中であらかじめ定められている用語（いわゆる予約語）です。

たとえば JavaScript の言語仕様がある日、「この予約語 function を fn に変える」となった時のことを考えると、エディタなどでテキスト一括置換を使って function を fn に入れ替えればすぐに対応できることが想像できるだろう。一方で、ある日「演算子*は整数同士の掛け算を表すもので、小数点同士の掛け算を表すには*.という演算子を使用しなければエラーとする」という仕様変更の場合はどうだろうか[*1]。プログラムを修正しようと思ったら実際にその関数が小数点の計算に使われているのか整数の計算に使われているのかを判断して置換しなければいけないので、先ほどのように一括置換というわけには行かない。

前者のように、プログラムの意味論そのものに影響を与えない言語仕様をシンタックス、後者のように意味論に影響を与えるものをセマンティクスと考えて貰えば良い。もちろんシンタックスも、たとえ意味論に影響は与えなくとも、予約語が短ければタイピングが早くなってプログラムが早く書けるかもしれない、というように言語の使い勝手に大きく影響する。

5.2.1 形式的定義と実装による定義

そもそもプログラミング言語を作るとはなんなのかを考えると、観念的には大きく分けて 2 つの作業が存在している。1 つは言語仕様を定義する作業であり、もう 1 つは仕様に従って実際にその言語を読み込み、実行できるプログラムを実装する作業である。ところが、現実的には（特に特定のドメインに特化した言語であれば尚のこと）言語の仕様というのは厳密に定義されているとは限らず、それを実行できるプログラムのみが存在していて、実装が唯一の言語仕様を表すものになっているケースも少なくない。

特に、C 言語のような低レイヤを記述する言語においては、何かセキュリティ関連の重大なインシデントがプログラムの予想外の動作によって引き起こされた場合、それはプログラマが言語仕様で未定義動作になっているコードを書いていたり、コンパイラが言語仕様に沿っていない実装をしていたり、言語仕様そのものの定義が甘いせいで、正しく言語仕様に沿ってプログラムを書き、コンパイラも言語仕様に沿ってコンパイルしたが予期しない動作が発生してしまったりなど、複数のケースが考えられ問題の原因と責任がどこにあるのかをはっきりさせることができるといった利点が考えられる。

音楽プログラミング言語においては、何かしら予期しない動作が起きたとしてもせいぜいプログラムがクラッシュする程度の問題にしかならないことに加えて、音楽のための言語に必須ともいえる、オーディオ入出力のようなペリフェラルを形式的に定義することはあまり一般的でないこと、また言語自体が特定の開発、実行環境（いわゆる統合開発環境：IDE）とセットになって設計されているため、形式的に言語仕様を定義するよりも、その実装におけるコメントや、ドキュメンテーションの充実で言語仕様らしい情報を補完する方が現

[*1] この整数と小数点の変数に対して算術演算子を使い分ける仕様は実際に OCaml に代表される ML 系言語で用いられている。もっともその目的は静的型付け言語においてユーザーが整数と小数点の計算を混同しないようにすることもあるが、そうすることで単に実装を簡略化できるという理由もある。

実的であるため、言語仕様＝唯一の実装となっているものが大多数である。

ただし、テキストベースの言語において、シンタックスを形式的に定義するには、バッカス・ナウア記法に代表される文脈自由文法による表記法などがあるのだが、これに似せた記法でシンタックスを定義するとテキストから抽象構文木という中間データ構造へ変換する、パースするためのプログラム（やそのための C 言語などのソースコード）を出力してくれる、コンパイラ・コンパイラと呼ばれるツールが存在しており、こうしたツールを用いれば実装による仕様定義ではあるもののある程度形式的定義風味にしてくれる。実際コンパイラコンパイラの代表的ツールである bison(yacc というツールの拡張版) は SUperCollider、ChucK、Faust といった多くの言語の実装で用いられており、筆者の実装した mimium もこれを用いている。

また音楽向け言語においてセマンティクスを形式的に定義する方法、形式的意味論での定義を用いている代表的な例としては Faust が挙げられる。形式的意味論は大きく分けて操作的意味論 (Operational Semantics) による定義と表示的意味論 (Denotational Semantics) による定義の 2 種類がある[*2]。なるべく簡単に説明すると、操作的意味論はプログラムがある装置（なんらかの仮想機械）上でどのように実行されるかの定義を示すことによって意味を決定し、それに対して表示的意味論はある項を別の言語の項へと変換する規則を定めることによって意味を定めるものである。Faust は形式的意味論を採用しており、ソースコードから、Block Diagram Algebra という、入出力を持ったシグナルプロセッサを並列、直列、分岐、合流、再帰という 5 種類の演算子で組み合わせで表現する体系へと変換し、それを C や C++ をはじめとした様々な低次の言語へと変換する (Orlarey, Fober, and Letz 2004)。

5.3 ドメイン固有言語のデザインパターン

次に、プログラミング言語一般の言語仕様の定義方法の話から、特定の応用領域（ドメイン）に向けたプログラミング言語、いわゆるドメイン固有言語（Domain-Specific-Language:DSL）の実装方法を概観しよう。

DSL は音楽に限らず、例えば Processing や GLSL のようなグラフィック生成のための言語であったり、Arduino や HDL のようなハードウェアの操作に特化した言語が存在する。先ほど挙げた bison のようなコンパイラ・コンパイラも、バッカス・ナウア記法風に書かれたテキストデータからパースプログラムのソースコードを出力する、いわばプログラミング言語実装のための DSL である。

一般的な汎用プログラミング言語は簡単に言えば、テキストという文字列データを解析して構文木と呼ばれるデータ構造に変換し、それを直接評価することで実行したり（インタプリタ型)、より低次の中間構造、例えば仮想機械への命令列や C 言語のような汎用言語のソースコードなどへの段階的な変換を経て実行される場合があるのだが、基本的な構造はテキストというデータを次々と機械語へと近い形のデータ構造へと変換し続けるパイプラインのような構造を取っていることが多い。

ところが、先ほどあげた Processing や Arduino といったツールは、実装の方法からすると Java や C++ といった汎用言語上に構築されたライブラリとほとんど見分けがつかない構造を持っているが、DSL に分類されることもある。

音楽のためのプログラミング言語も同様に、汎用プログラミング言語のライブラリ的に実装されているものと、構文解析を 1 から行うもの、また両者をハイブリッド的に用いるものなど複数のアプローチが存在している。構文解析から行うものを External DSL、ライブラリのような形で実装するものを Internal DSL と呼ぶこともある (Nishino and Nakatsu 2015, p28)。

このような External,Internal といった実装のアプローチの違いを Spinellis はデザインパターンとして提示した。そのうち、実装方針に関わる一部を以下に列挙する (Spinellis 2001)。

- 文字列解析

[*2] 厳密には公理的意味論（Axiomatic Semantics）なども存在するが、汎用言語でも定義に用いられている例は少なく、音楽プログラミング言語で使われている例は知る限り存在しないので本稿では省略する。(**UnderstandingComputation**) を参照。

- 言語特殊化
- 言語拡張
- Piggyback(おんぶ)
- ソース to ソース変換
- パイプライン

文字列解析は、すでに説明した、テキストデータを直接読み込んで抽象構文木のようなデータ構造へ変換するプログラムを書く方法だ。これは当然、文法の設計の自由度が一番高い代わりに、構文解析という比較的複雑になりやすいプログラムを自ら実装する必要がある。しかし、特定の応用領域があるとは言っても、必ずしも表面的な文法をゼロから設計しなくても良い、むしろ、汎用プログラミングにもある程度触れたことがあり、そうした言語の記法からかけ離れない方が習得しやすいというケースもあるだろう。そうした時には言語特殊化、あるいは言語拡張という既存の言語に乗っかる形でのライブラリ的実装をするという選択肢がある。

言語特殊化というのは、たくさんある言語機能のうちの敢えて一部だけを使うような構造をとる言語である。先程の例で言えば、Processing や Arduino がそれに当たる。Processing は画面上に四角形を描画する `rect(100,100,200,200)` 関数のように、画像描画のための関数を呼び出せるが、このホスト言語（ライブラリ実装に用いているプログラミング言語）は Java という言語であり、関数定義などの意味論に関わる文法は Java のものに準ずる。通常、特定の目的に特化した関数は名前空間のような、命名規則でなく言語仕様に存在する構造化のための機能を用いることが多いが、敢えてそのような機能を使わず、はじめてプログラミングを行った人間でも限られた表記の規則だけ覚えれば実行することができる、という状態を実現しているといえよう。

こうしたライブラリとしての DSL を実装するもっとも大きな利点は、ホスト言語上で実装されているライブラリの資産を活用できる点である。例えば、Processing の中で音声信号処理のライブラリを用いたければ、Java 言語のために実装されたライブラリをそのまま使うことも可能である。もちろん、そのライブラリ内ではいかに Java のより高度な言語仕様が用いられていても、問題なく実行できる。

言語拡張とは、言語制限と同様ホスト言語上のライブラリとしての実装をする点では共通しているが、ホスト言語自体に意味論を自己拡張できるような機能が存在していると、ライブラリでありながらまるで別の言語を用いているかのように DSL を実装することが可能になる。これは次のセクションで詳しく説明しよう。

Piggyback(おんぶ) とは、構文解析を行うような DSL とライブラリとしての DSL の中間的アプローチとも言える方法だ。例えば、bison というコンパイラコンパイラは、その構文の使用自体は独自に定義されているものの、文を解析した後に行う処理、アクションと呼ばれる部分の記述は、後に変換される C 言語や C++ 言語の記法をそのまま用いることができる。このような別の言語仕様をある言語上に埋め込むようなパターンが Piggyback だ。このアプローチを用いることで、言語の構文の自由度を高めつつも、部分的には別の言語を借りて実装のコストを減らすことができる。

ソース to ソース変換は、ある言語のソースコードを別のホスト言語へ変換する方法のことだ。Piggyback とどう違うのかというと、例えば典型的なソース変換を行う言語である Faust は自身の言語内で意味体系が完結している。そのため、元々は C++ のソースコードに変換する機能がメインの言語であるものの、現在は C 言語、Rust、D 言語、Java、LLVM IR などさまざまな言語に変換することができるようになっています。このアプローチの利点は 1 つのホスト言語に縛られないながらもホスト言語周りのインフラストラクチャ（ビルド、デプロイのためのツールとか）を活用することができる点だと言える。

5.3.1 言語拡張と自己反映性：CoffeeCollider を例に

ライブラリとしての DSL 実装の中でも、ホスト言語の機能が拡張性がある場合はかなり構文の自由度がある言語設計を行うことが可能だ。

その例を、mohayonao による CoffeeCollider という DSL を例にして見ていこう。

CoffeeCollider は、音楽プログラミング言語の中でも代表的な言語である SuperCollider の構文を模した、CoffeeScript という Web ブラウザ上で動作する言語のライブラリとしての DSL である。なお、CoffeeScript 自体はブラウザ上で動作する唯一の言語である JavaScript へと変換されることで動作する、いわゆる AltJS と呼ばれる言語の一つだ (**mohayonao**)。SuperCollider は構文解析から行う言語であり、ネイティブアプリケーションとして動作するものの Web ブラウザでは動作しない。その代わりに CoffeeCollider は SuperCollider 上における Unit Generator の接続を Web Audio API という、Web ブラウザに近年標準的に組み込まれるようになったインターフェースを用いることで実現している。

CoffeeCollider は文字列解析を自分では行っておらず、特定のオブジェクトに対する + 演算子や*演算子の挙動をオーバーロードすることで、CoffeeScript の文法で可能な範囲の表現でシンタックスを SuperCollider へと模している。以下に CoffeeCollider のサンプルコードを示す。

```
(->
  noise = PinkNoise.ar(0.2)
  noise = Mix Array.fill 10, (i)->
    Resonz.ar(noise, i * 800 + 200, 0.05)
  noise = (noise * 0.2 + noise * Decay.kr(Dust.kr(0.5), 10))
  noise = RHPF.ar(noise, LFNoise0.kr(0.5).range(220, 880), rq:0.001)
  CombL.ar(noise, delaytime:0.5, decaytime:25).dup() * 0.5
).play()
```

この Example を SuperCollider 本来の記法で書いたのが次のコードだ。

```
{
   var noise = PinkNoise.ar(0.2);
   noise = Mix.new(Array.fill(10, {arg i;
     Resonz.ar(noise, i * 800 + 200, 0.05)}));
   noise = (noise * 0.2 + noise * Decay.kr(Dust.kr(0.5),10));
   noise = RHPF.ar(noise, LFNoise0.kr(0.5).range(220, 880), rq:0.001);
   CombL.ar(noise, delaytime:0.5, decaytime:25).dup() * 0.5
}.play;
```

SuperCollider での {} で囲むことでオブジェクトを生成するシンタックスを、CoffeeScript における無名関数 (`-> statements`) を利用して似せていることがわかる。

さて、coffeecollider の特徴的な点としては、**演算子のオーバーロード**を積極的に活用することでテキストをパースするプログラムを書くことなく、CoffeeScript そのままで記法を可能な限り SuperCollider に近づけているという点だ。

演算子のオーバーロードとは、たとえば + や*などの二項演算を数値の加算乗算やテキストの結合などの言語組み込みの型だけではなく、自身の定義したカスタム型に対して新しく振る舞いを定義してやれるような機能のことだ。

たとえば、Number 型 2 つで構成されるの 2 次元ベクトルの型を定義したら + 演算子を使ったら要素同士をそれぞれ加算できたり、*演算子を使ったら直積、内積や外積を計算できるようにしたらコードの記述が短くできたり、一般的に数学で使われるような記法と近づけられることでソースコードを直感的に読めるようにできる。

演算子のオーバーロードができるようになっている汎用的な言語としては、たとえば C++、Scala、Haskell などがある。特に Haskell では、あらゆる関数を中置演算子のように使える上に`!` `#` `$` `%` `&` `*` `+` `.` `/` `<` `=` `>` `?` `@` `\` `^` `|` `-` `~`などの文字から任意の文字を組み合わせて演算子を作ることもでき、`<=>??@`演算子のような独自演算子を作ることもできてしまう。

例えば Javascript には演算子オーバーロードの機能がないので、Tone.js(**tonejs**) といった JavaScript 上での音声処理ライブラリでは、信号処理プロセッサのようなユーザーによって定義されたデータ型同士を組み合わせるためには関数呼び出しやオブジェクトのメソッド呼び出しといった記法を用いて表現するしかなくなる。

一方、CoffeeScript には演算子オーバーロードがついているので SuperCollider のような、全く別の言語体系の言語であってもある程度表面上のシンタックスを近づけることができる。

演算子オーバーロード以外に、こうした言語自体の記法を自己拡張するような機能は他にも、Kotlin の Type-Safe Builder や F#の Applicative Computation Expression などがある。

またマクロのような、言語本体の処理より前にテキストや Abstract Syntax Tree をユーザーが定義したルールを用いて置き換える記法も構文拡張の一つとして見ることができる。

こうした言語の意味を拡張していくような言語機能は、場合によっては自然な記述を可能にする一方、場合によっては同じ言語で書かれているのにソースコードの見た目が全く異なってしまい、かえって読みにくくしてしまうような効果も持つ。そのため、言語設計としては go 言語のように意図してオーバーロードを禁止することで、ソースの見た目の一貫性をキープする方針を取る言語も少なくない。その意味では LISP の系列の言語では、Syntax を敢えて S 式と呼ばれる括弧で括る記法に統一してしまうことによって、マクロなどで機能を拡張し続けても、結果的にソースコードの見た目は一貫性を保つことができる。

極端な言語の一例としては、コンパイラの挙動を実行コード側からかなり自由に変更できるようにすることで、ライブラリや利用目的ごとに言語のシンタックスを大きく変えられてしまう Racket(Culpepper et al. 2019) がある。Racket そのものは本来 S 式の言語だが、*#langslideshow* のようなシンタックス指定の命令を記述することによってその行以下で利用する言語をスライドショーを作る機能に特化した言語に切り替えられたり、ライブラリを切り替える感覚で言語の表面的な見た目までを切り替えられてしまう。この設計思想は、解きたい問題のドメインに合わせてまずそのドメインに特化した言語を作る、Language-Oriented Programming と呼ばれている。

実装面から見れば、マクロにしても Racket のような高度なコンパイラの挙動変更にしても、ソースコードをコンパイルしながら適宜その内容によってコンパイラ自体も変更されるというフィードバックプロセスが、言語の自由度を高めるほどに複雑化していくため、単に入力されたテキストデータを低レベルのデータまで変換していくパイプ的な構造から遠ざかっていくことになり、実装の難易度も一般的に上がっていく。

5.3.2 TidalCycles - ハイブリッドなアプローチ

また TidalCycles(McLean 2014) のように文字列解析、言語拡張、言語特殊化を組み合わせて実装されることもある。

```
d1 $ sound "bd*4" # gain (every 3 (rev) $ "1 0.8 0.5 0.7")
```

この 1 行において、$や#はホスト言語である Haskell における中置演算子の機能をオーバーロード（**言語拡張**）したものである。($は括弧で関数実行の順番と単位を切り替えるのに近い役割をする、Haskell 標準の機能そのままだ)。sound や gain、every などは Haskell 上で TidalCycles ライブラリとして定義された関数の名前である。また同時に、ユーザーは Haskell の関数定義や型宣言など高度な文法を知る必要はない（**言語特殊化**)。

そのため TidalCycles はおおむねライブラリとしての DSL の性質を備えている一方、“ “で囲まれた部分は Haskell 上ではただの文字列であり、Parsec という Haskell の**字句解析**ライブラリで内部的にパースしている。この記法は Bol Processor という表記法をベースに独自に定義されたものである。

また最終的に TidalCycles は SuperCollider という音声合成エンジンに対して OSC(Open Sound Control) というネットワークコマンドを送ることで音を出している (**パイプライン**)。

5.3.3　DSL の中でも、音楽特有の問題 - なぜライブラリとしての DSL ではダメなのか？

Haskell みたいな DSL 作りやすい言語で全てライブラリとして実装しちゃダメなのか？→基本的には厳しい
OS のプリエンプティブスケジューリング（ユーザプログラムが時間に関与できない）
例外- Xenomai の様なカーネル拡張
信号処理中にヒープメモリ確保をできない問題

- テキストをパースするプログラムを書かなくてもライブラリでもホスト言語が演算子オーバーロードのように自己拡張する仕様を持つ場合 DSL としての性質を帯びさせることができる。
- パースするプログラムをゼロから書いた方が言語設計の自由度は高まる一方で、実装のコストが高くなる。さらにユーザーとしてもまったく新しい言語を 1 から勉強することになるので学習コストが高い。

McCartney は SuperCollider の設計についての論文で、究極的には音楽のために言語をゼロから開発する必要はないと述べる。

> コンピュータ音楽に特化した言語は実際必要なのだろうか？少なくとも理論的には、私はそうではないと思う。今日の汎用プログラミング言語で利用できる抽象化方法は、コンピュータ音楽を便利に表現するフレームワークを構築するのには十分なものだ。ただ残念なことに、実用的には、今すぐ使える言語の実装としては欠けている部分もある。多くの場合、ガベージコレクションはリアルタイムでは実行されず、引数の受け渡しはあまり柔軟でない。遅延評価がないと、パターンやストリームの実装がより複雑になる。SuperCollider を書いたのは、コンピュータ音楽のために柔軟に使えるような抽象化方法を持たせたいと思ったからだ。将来的には他の言語の方が適しているかもしれない。SC Server の合成エンジンと言語を分離する目的の一つは、SuperCollider 言語とクラスライブラリで表現されている概念を他の言語で実装することを模索可能にすることだ。他にも、将来的にコンピュータ音楽のために面白い可能性を秘めた言語として、OCaml (www.ocaml.org)、Dylan (www.gwydiondylan.org)、GOO (www.googoogaga.org)、そして、偶然にも文法規則の多くに似た仕様を持つスクリプト言語である Ruby (www.ruby-lang.org) がある。(McCartney 2002)

実際、後半で述べられているように SuperCollider という言語そのものを積極的に利用する人は少なくなった一方、音声合成エンジンとしての SCServer はさまざまな形で利用され、音楽言語のインフラストラクチャとして機能している。FoxDot は Python、Overtone は Clojure、TidalCycles は Haskell、Sonic Pi は Ruby と Erlang) といったように実際に多様なホスト言語から利用されている。

しかしこの論文から 20 年近く経過した現在でも、汎用言語のライブラリとして音楽プログラミング言語を実装するには未だ限界があると言わざるを得ない。

Elementary(JS+JUCE) を例に

Ladder of Exprerience(加藤) High Ceiling、Lower floor

5.3.4 ビジュアル言語のシンタックスと保存フォーマット

一般的には中間表現を低次へ変換していくほどより抽象的な操作に近づいていくので言語処理系の実装が汎用言語の実装に近くなっていく。

また Max や Puredata のようなビジュアル言語の場合は、保存されているテキスト/バイナリデータを解釈してビジュアルインターフェースとして表示する評価と、音声処理のためのデータ構造としての評価と 2 方向の評価が行われていると解釈することができる。この時の保存されているデータ形式は例えば Max では JSON という Javascript のための汎用データ表現形式を用いているように、その言語の Semantics とは関係のない汎用的なデータ構造を利用している場合もある。

5.4 音楽言語設計におけるトレードオフー General,Efficient and Expressive

少ない例としては、Mcpherson らは音楽プログラミング環境の設計者へのインタビューを通じて、言語設計者がそれぞれどのような価値観を持って機能実装をしているかを分析している (McPherson and Tahlroğlu 2020)。

(Mcpherson の例を解説)

音楽言語の実装に関する論文では、General、Efficient、Expressive という 3 つの語がよく使われる用語でありながら、実際には指している意味が複数混在しているので用語を整理する必要がある。

例えば、Brandt は音楽言語設計における大きな問題として Expressiveness と Generality の Tradeoff が存在するとして、それらの両立を目指して関数型言語 OCaml 上でのライブラリ Chronic を開発している (Brandt 2002)。

Figure 1.2: The expressiveness vs. generality tradeoff

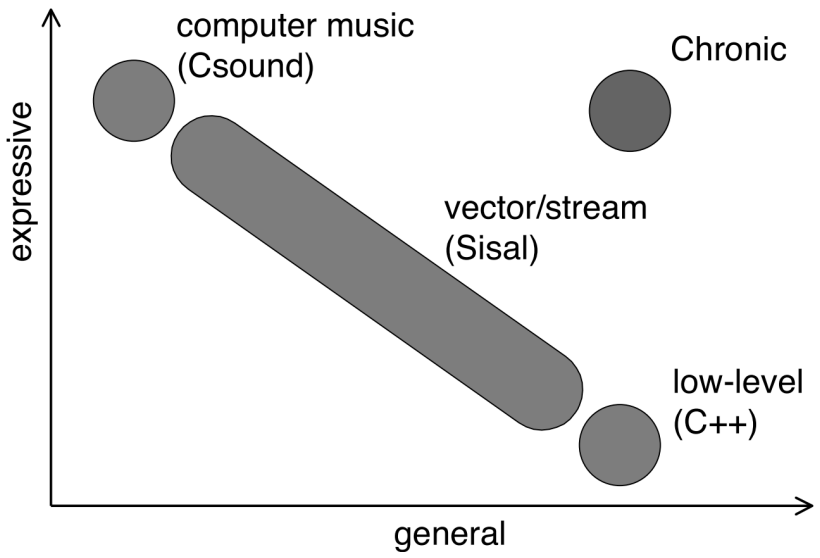


図 5.1: image-20210606155655532

ここでの Expressiveness とは “ユーザーが読みやすく書きやすい”、General とは “可能な表現の範囲が広い” という意味で用いられている。また Chronic は OCaml というホスト言語の実行速度の問題もあって、リアルタイムで実行はできなかった。つまり Chronic では general と expressive の両立はできたが実行時パフォーマンスという新たなトレードオフが発生していたことがわかる。

また Lazzarini は CSound の設計において、Score、Orchestra、Instrument という 3 種類の内部言語を用いるアプローチにおいて、

> 音楽プログラミングにおいて多言語アプローチを取ると、ユーザは**汎用性 (Generality)** と**効率性 (Efficiency)** のバランスを取りやすくなる、

と述べている。ここでの用語は、続く以下の説明を参照すると、

> タスクに応じてプログラマは異なる複雑度のエントリーポイントを選択することが可能になる。より低く汎用的なレベルではより複雑なコードの設計が必要だが幅広い結果を得られる。一方、より高い、特殊化、具体化されたレベルではプログラミングに必要な労力という観点からプロセスはより効率的になるだろう。(Lazzarini 2013)

つまりここでの Efficiency は、実行コストのことではなく、ユーザーが目的のプログラムを構築するまでの効率のことを指している。

また Dannenberg のサーベイでは。

> This article considers models of computation that are especially important for music programming, how these models are supported in programming languages, and how this leads to **expressive** and **efficient** programs.(Dannenberg 2018)

ここでの Efficient は programs に掛かっているので Runtime Cost のことを指していると推察できる。

このように、同じ Efficiency というワードでも Computer が負担するコストとユーザーが負担するコストという異なる事項を指している場合があることがわかる。

汎用のプログラミング言語においては Coblenz らが主に以下のような評価のための用語整理とその意味の明確化を試みている (Coblenz et al. 2018)。

- General
- Efficient “Execution cost”
- Expressive “To what extent can users specify their intent using the formal mechanisms of the language?”
- Modifiability “How easy or hard is it to adapt software to changing requirements?”
- Learnability
- Understandability
- Portability
- ...

そこで、次の項では Mcpherson らや Coblenz らの分類を参考にしながら、改めて評価のための語彙を整理する。そのためにまずは、音楽プログラミング言語を評価、また設計の指針となるための value を、ユーザーがソースを編集し、実行し、そのフィードバックが帰ってくる human-in-the-loop システムとしてモデル化し、その際に考えられるコストや自由度をユーザー、コンピューターそれぞれの観点で分類するという方針をとる。

5.5 音楽プログラミング行為のモデル化と評価語彙の提示

Human in the Loop なシステムとしての音楽プログラミングモデルの行為 (Anderson and Kuivila 1990)

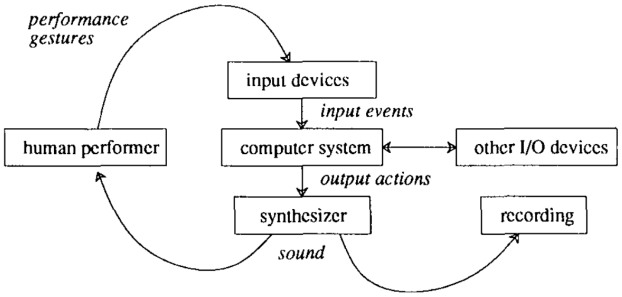


図 5.2: Anderson と Kuibila による音楽プログラミングのモデル。

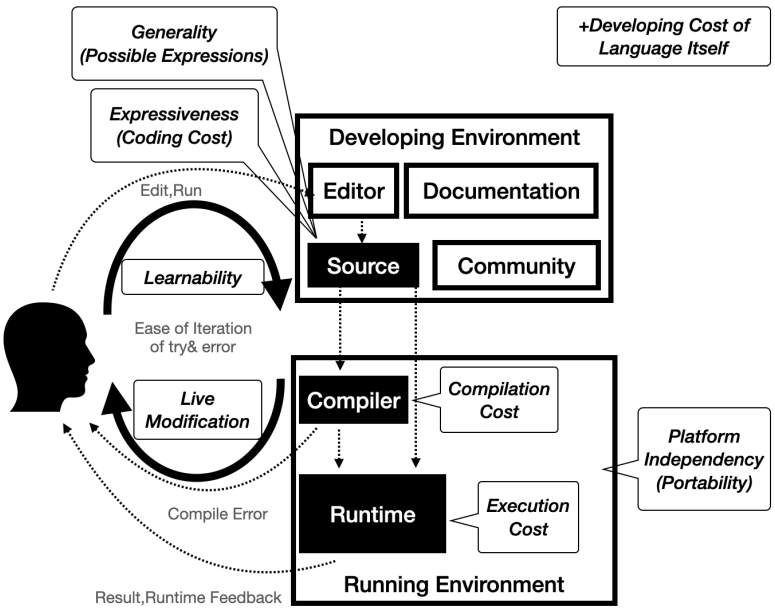


図 5.3: Screen Shot 2021-06-07 at 15.12.26

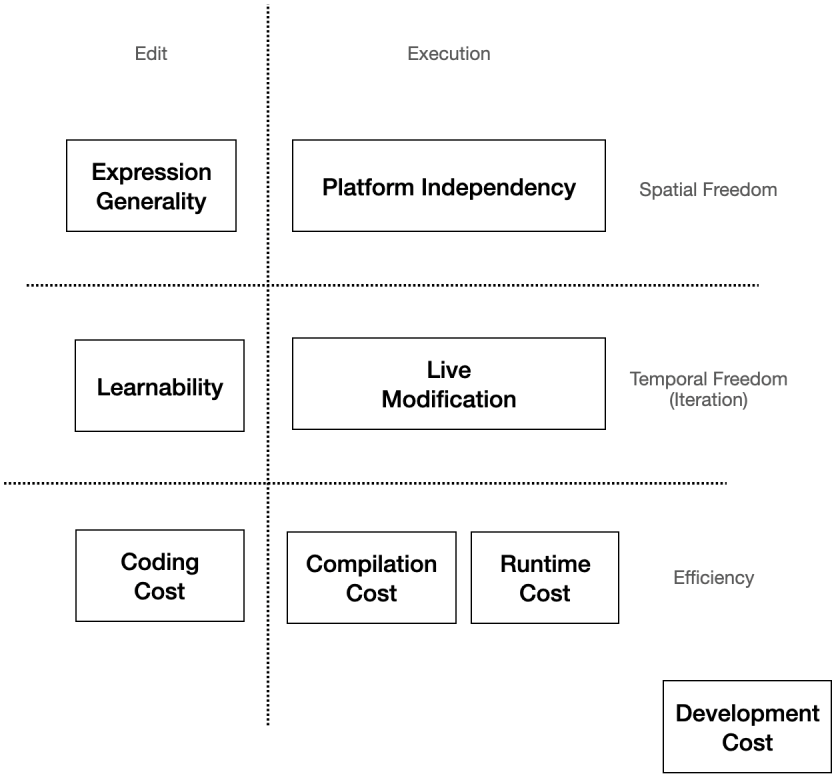


図 5.4: Screen Shot 2021-06-07 at 16.25.25

ユーザーへのフィードバックが音を出す前にわかる場合と、音を鳴らして初めてわかるものとある（型チェックとか）

5.5.1　User-Side

学習しやすさ

表現可能な空間の広さ (Generality) が Mcpherson2020 における Open-Endedness、Brandt における Generality にあたるものと言える。

ユーザーが必要とするコーディングコスト、すなわちユーザーが想定している表現にどれだけ簡単に辿り着けるような言語仕様になっているかどうかが Expressiveness にあたるものと言える。

5.5.2　Computer-Side

Compilation cost

コンピューターが必要とするコスト（Runtime Efficiency/Execution Cost）

実行可能な空間の広さ（Portability）

5.5.3　Edit-Execute の繰り返しやすさ

ユーザー側：Learnability

コンピューター側：Dynamic Modification

(Mcpherson2020 における Rapid-Prototyping、Mcpherson2020 の Dynamism は実行中に動的に UGen のデータフローを作り替えられるかどうかという話なので少し違う)　ここが極まると Live Coding になる

Development 自体のしやすさ

C++ とかで作られてると大変、インタプリタ型なら簡単

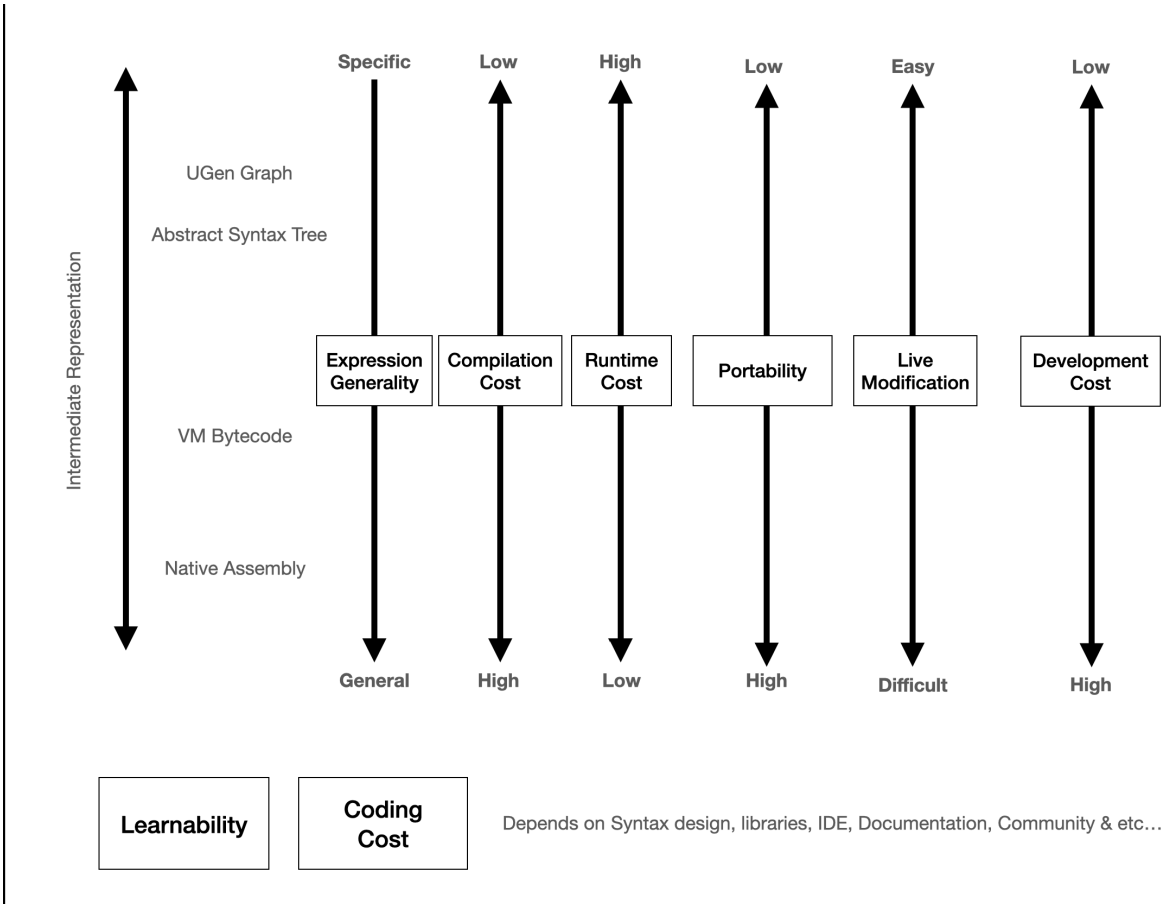
言語の自己反映性が高ければ言語自体のコーディングしやすさと一致してくる（機能拡張がライブラリを書くことで可能になる）

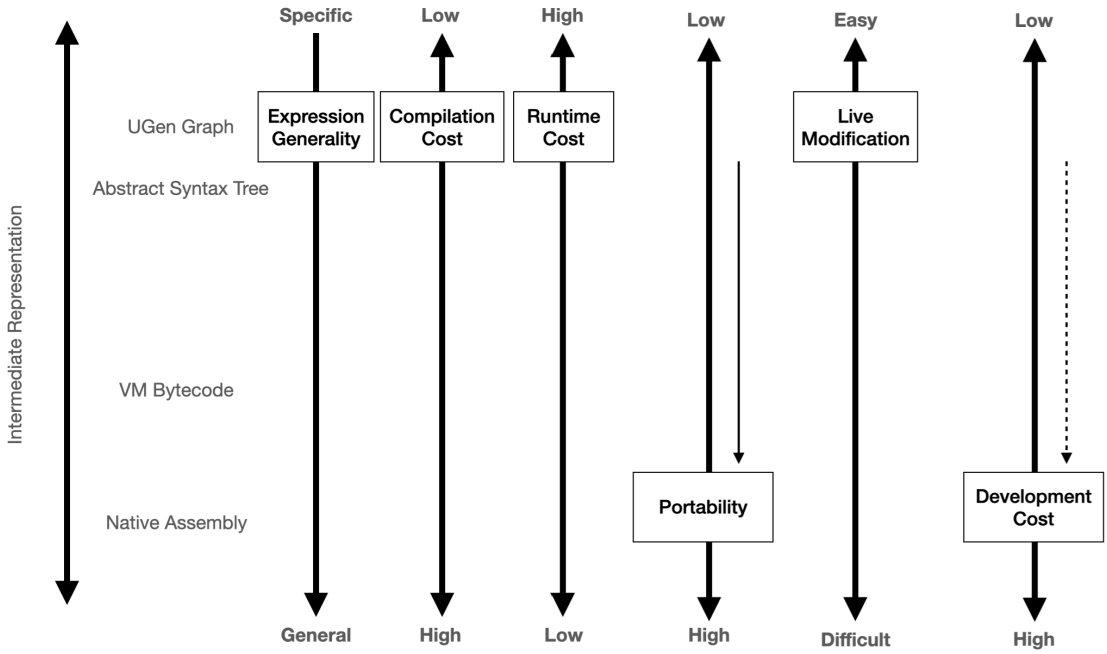
5.5.4　それぞれのトレードオフ

10 個全部を満たすのは無理、少なくとも全部を撮ろうとすると Development が超大変

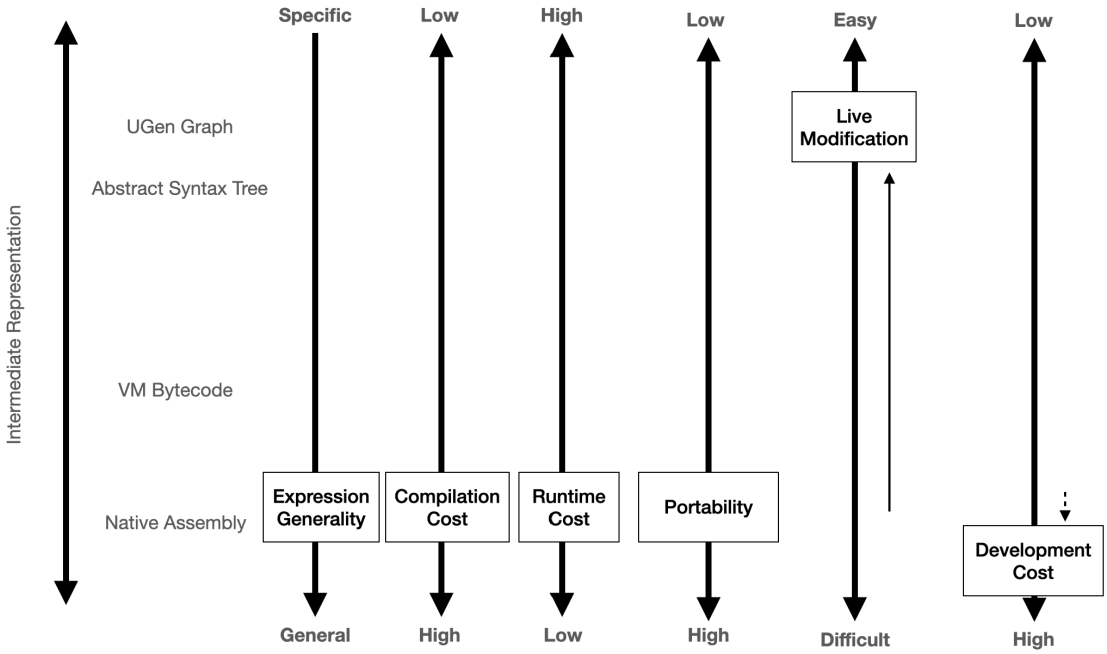
他、具体的には（ここを綺麗に整理できたら嬉しい）

- インタプリタ型の実装だと Edit-Execute の変更はしやすくなる、その代わり信号処理とかをスクラッチで実装するのは実行コストが嵩む
-





抽象度の高い中間表現を他プラットフォームで動かそうとすると、各プラットフォームごとでの実装の手間が増える
＝Development Costは増大



低レイヤまで変換するタイプの言語にライブコーディングの機能を持たせようとすると、JITコンパイルを行う必要があるので
Development Costは更に増大

5.6 Multi-Language Paradigm の

5.7 小括

本章では音楽プログラミング言語を主に設計、実装の視点から検討し、音楽プログラミングという行為と計算機内で発生するプロセスを Human-in-the-Loop モデル化し提示することでを改めて整理した。その上で音楽のための言語の特性や、評価するための語彙について検討した。

まず音楽プログラミング言語を実行するソフトウェアとは、通常のプログラミング言語と同じようにテキストデータがコンパイラによって様々な中間表現を経てより低次なデータ構造へと変換され、ランタイムへと渡されるという大まかな構造を持っている。

この時、実装方法としてはテキストデータを直接解析せずとも、あるプログラミング言語の自己反映性 (演算子オーバーロードやマクロ) が高ければ、その言語自体の機能を変化させる形でライブラリとして言語を実装することもできる。ただし、多くの自己反映性の高い言語は、リアルタイムでの動作をあまり考慮しないメモリ自動解放機能のような、信号処理を行う上では障害となる言語機能を同時に備えおり、楽譜レベルの記述やリズムパターンのような比較的抽象度が高く、発生する時間間隔が大きい表現の記述にその対象が絞られるという問題点がある。

そして、中間表現の粒度を Unit Generator のグラフ構造、仮想機械のバイトコード、Block Diagram Algebra、LLVM IR というようにより機械語へ近いレベルへ下げていくと、一般的に表現可能な範囲が広まり、実行性能が良くなるといった利点があるが、コンパイルの過程がより汎用プログラミング言語のそれに近づき複雑化する、コードのプラットフォームをまだいだ可搬性が下がる、コードを動的に部分変更することが難しくなるといった根本的トレードオフが存在する。

このトレードオフを両立することは不可能ではないが、必然的に言語の構造そのものが複雑化することに繋がる他、その言語の実装過程そのものに音楽的要素が無 kl うなることから、開発者にとってはドメイン固有言語を実装モチベーションを下げることにも繋がるという問題があった。

第 6 章

音楽プログラミング言語 mimium の設計と実装

6.1 mimium の設計

本章では、第 3,4 章で説明した背景をもとに開発したプログラミング言語 mimium(**mi**nimal-**m**usical-med**ium**)[*1]を設計した。の具体的な実装について記す。

第 3 章では、CSound の Score-Orchestra-Instrument に代表される、Multi-Language Paradigm の功罪について議論した。対象とする表現に絞って言語をそれぞれ開発することは、関心を分離する一方で、既に存在している音楽の様式そのものを言語仕様に組み込んでしまうことでその外側の表現を探索することが難しくなる問題と、作曲を主に行う人が信号処理のある部分について少しパーソナライズした変更をしたいと思った時のような、横道へと逸れることを難しくしてしまうという問題があった。

仮に同じ言語の Semantics 上で、ライブラリとして信号処理にしても、楽譜レベルの信号処理にしても表現できるような言語体系を作ることができたなら、既存の音楽様式にロックインされない表現の可能性を高めることができるだろう。

第 4 章では音楽プログラミング言語の実装方法の観点から、中間表現の粒度とそれに伴う可能な表現の範囲、実行コスト、コンパイルコスト、動的変更のしやすさ、開発コストなどの要素にトレードオフの関係があることを示した。

6.1.1 既存の言語との差分

こうした、目的を音楽に特化させつつもなるべく広い表現を保証した言語という視点で、既存の言語の仕様とその不足点について検討してみよう。

表 6.1 は mimium と既存の言語の仕様を比較したものである。mimium は時間方向に離散的な制御の記述と、信号処理と 1 つのセマンティクスで実現し、信号処理の実行速度は JIT(実行時) コンパイルを用いることで C++ などの低レベル言語での記述と同等にしている。またユーザーはメモリーの確保、開放のようなハードウェアの意識をする必要がない。

6.2 mimium の設計

本セクションでは、mimium の言語仕様と実行環境の詳細を記述する。

まず、基本的な構文を紹介し、メモリ管理などのハードウェアを意識する必要がないこと、型推論により変

[*1] 開発時点でのバージョンは 0.4.0 をもとに解説している。ソースコードは https://github.com/mimium-org/mimium で公開されている。

表 6.1: mimium と既存の言語の仕様比較。それぞれの要素が 4 章の目標とどう対応づけられるのか追記したい。

	Pd/SC	ChucK	Extempore	Faust&Vult	Kronos	*mimium*
スケジューラ	○	○	○	-	○	○
サンプル精度のスケジューリング	-	○	○	-	○	○
低レベルな UGen の定義	-	○	○	○	○	○
DSP コードの JIT コンパイル	-	-	○	○	○	○
UGen 内部状態の関数型表現	-	-	-	○	○	○

数の型アノテーションを省略できることを示します。次に、mimium の実行環境（コンパイラとランタイム）の一般的なアーキテクチャを紹介し、信号処理であっても、LLVM を介してすぐにネイティブバイナリにコンパイルして実行することができることを示す。

さらに、連続的な信号処理と離散的な制御処理を統一されたセマンティクスで記述することを可能にする、mimium の 2 つの特徴的な機能について説明する。一つ目は、サンプルレベルでの決定論的なタスクスケジューリングのためのシンタックスと、スケジューラーの実装です。2 つ目は、信号処理用の UGen を言語上で定義するために用いられるセマンティクスとそのコンパイルプロセスについて、関数と内部状態変数のペアのデータ構造という点で既存のパラダイムと比較しながら説明する。

議論のセクションでは、2 つの問題を取り上げている。(1)mimium は離散的な制御と信号処理を統一的なセマンティクスで記述できるが、離散的な処理の記述方法は命令的なものが多く、信号処理に用いられる関数パラダイムは互いに大きく異なること、(2) 現在の実装では、Faust や Kronos と異なり、信号処理のためのステートフル関数のパラメトリックな複製を表現できないこと。また、前述の問題を解決する方法として、多段計算を用いる可能性についても説明する。

6.2.1 基本的な文法

mimium の基本的なシンタックスは、Rust(Klabnik, Steve and Nichols 2020) をベースにした。

シンタックスの選択は常に**すでにある程度普及している言語体系**を眺めた上で設計せざるを得ない。Star らがインフラストラクチャは常に既存のインフラストラクチャの上に乗ることで成立すると指摘する状況ー例えば電話線のインフラストラクチャがすでに存在していた電信のインフラストラクチャに乗っかることで成立しているようなものと似ている。

Rust の構文は予約語が比較的短いので音楽のように素早くプロトタイピングを行う分野に適していることが主な理由である。また、既存の言語の構文と似せると、既存の言語のシンタックスハイライトを再利用しやすいという副次的効果も得られる。

コード 6.1 に基本的な言語仕様を示す。BNF による形式的な言語仕様定義は付録 A にて示した。変数の宣言は、関数のスコープ内で新しい名前の変数に何らかの値を代入することで自動的に行われます。変数を宣言する際には、コロンの後に型名を記述することで、値の型を明示的に指定することができます。型名が省略された場合は、文脈から型を推測することができます。データ型には、void（空の値の型）、numeric（整数、浮動小数点型の区別はなく、内部的にはデフォルトで 64bit-float）、string といったプリミティブな型のほか、function、tuple、array といった合成型がある。また、ユーザー定義の型エイリアスを宣言もできる。

基本的な構文には、関数定義、関数呼び出し、if-else 文を使った条件分岐などがある。また、関数型のパラダ

```
//コメントはダブルスラッシュ
//新しい名前の変数への代入が変数宣言になる
mynumber = 1000
//現在のところ全て変数はmutable
mynumber = 2000

//型注釈もできる
myvariable:float = 10
//型エイリアス
type FilterCoeffs = (float,float,float,float,float)

//文字列型は現在のところリテラルでのみ利用され、音声ファイル読み込みなどに用いられる
mystring = "somefile.wav"

//配列型
myarr = [1,2,3,4,5,6,7,8,9,10]
//配列アクセス
arr_content = myarr[0]
myarr[4] = 20 //配列書き込み

mytup = (1,2,3) //タプル型
one,two,three = mytup //タプルの展開

fn add(x,y){ //関数定義
  return x+y
}

add = |x,y|{ x+y }//無名関数

//で囲った複数の文をつの式として扱える{}1
z = { x = 1
        y = 2
            return x+y } //z should be 3.

//条件分岐と再帰関数
fn fact(input){
  if(input>0){
    return 1
  }else{
    return input * fact(input-1)
  }
}
// 文は式としても使えるif
fact = |input|{ if(input>0) 1 else input * fact(input-1) }
```

Listing 6.1: Basic syntax of *mimium*.

```
fn dsp(input:(float,float)) ->(float,float){
    left,right = input
    out = (left+right)/2
    return (out,out)
}
```

Listing 6.2: Example of *dsp* function which merges stereo inputs and returns the same signal to each output channels.

イムを取り入れ、if 文を直接値を返す式として使用することができます。これは、複数の文（代入構文や関数実行）を {} で囲み、最終行の *return* 式の値を式として返す構文（Block）を持つことで実現している（*return* を省略することも可能）。同様に、関数定義は、無名関数を変数に代入する構文のシンタックスシュガーとして定義されている。

mimium は静的型付け言語で、すべての変数と関数の型がコンパイル時に決定される。型推論（現在は単相）は Hindley-Milner のアルゴリズムに基づく。

また、DSP 処理の高速化のために、メモリの割り当てと解放はコンパイル時に静的に決定され、ランタイムではガベージコレクションが行われない。

6.3 *mimium* での信号処理

mimium では、*dsp* という関数をユーザーが定義すると、オーディオドライバーと音声の入出力に仕様される。例をコード 6.2 に示す、。この場合、*dsp* 関数の型は、1 つ以上の float 型からなるタプル型を取り、かつ、1 つ以上の float 型からなるタプル型を返す関数型にする必要がある。タプルの各要素が、オーディオドライバの入力チャンネルと出力チャンネルに対応する。例では、オーディオドライバから 2 つのチャンネルの入力を受け取り，それらをミックスして、左右それぞれのチャンネルに複製して返している。

mimium には組み込み関数として、基本的な算術演算、libc のmath.h で定義されている三角関数や指数関数などの数学関数、*mathitdelay* や *mathitmem*（1 サンプルの遅延）などの組み込みの状態を持つ関数 (後述)、libsndfile(Lopo 1990) を使って wav ファイルを読み込むための *mathitloadwav* 関数、デバッグのための *mathitprint* 関数などがある。フィルターやオシレーターは、これらの関数を組み合わせることで、すべてライブラリとして定義することになる。

6.4 アーキテクチャ

図 6.1 に *mimium* の実行環境（コンパイラとランタイム）のアーキテクチャを示す。

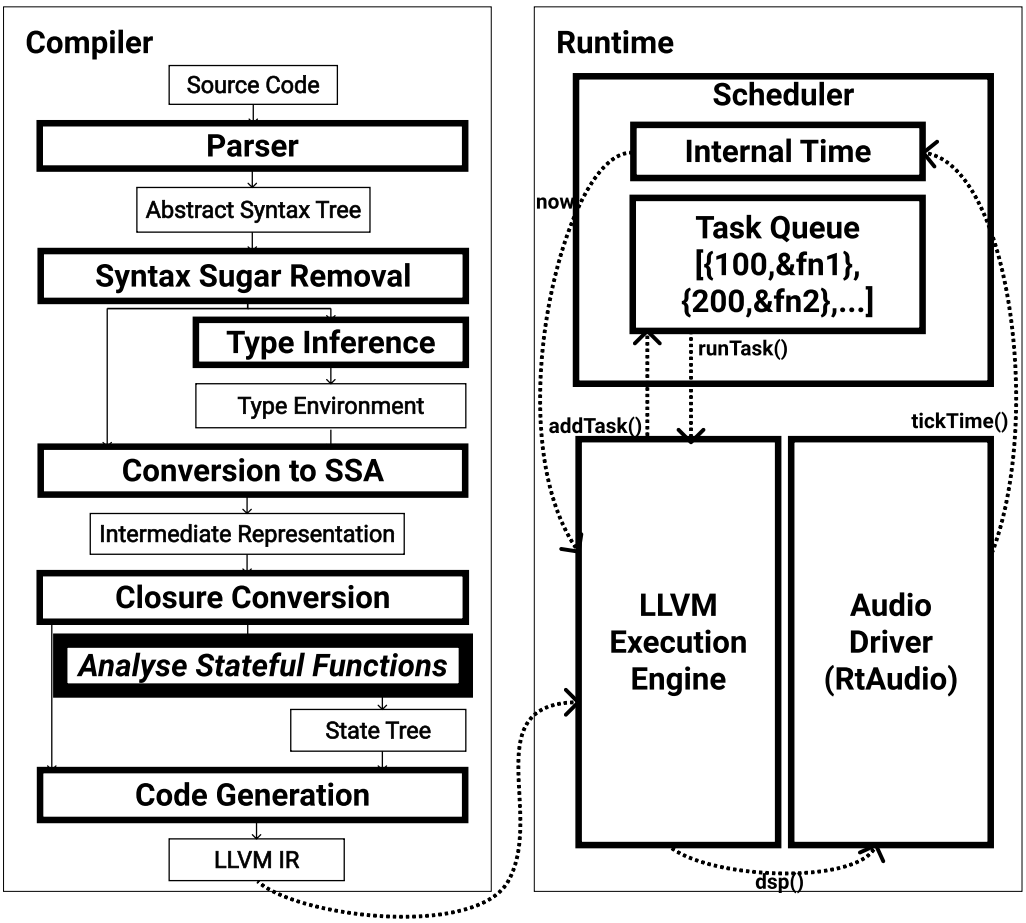


図 6.1: Architecture of *mimium* compiler and runtime.

コンパイラの構造は、一般的な関数型言語の構造に近く、mincaml(Sumii 2005) をベースに、C++ を用いて実装されている。

ソースコードのテキストデータは、まず抽象構文木（AST）に解析され、シンタックスシュガーを変換した後、AST はラムダ計算ベースの木構造に変換される。次に、型推論、型検査を行い、すべての変数の型を決定する。AST は、型情報とともに、すべての変数が一度だけ割り当てられる静的単一代入（SSA）形式の中間表現に変換されます。この段階ではまだ関数の入れ子定義が可能になっているので、関数定義から自由変数を取り除くためにクロージャ変換が行われる。

クロージャ変換と低レベルコード（LLVM IR）生成の間には、*mimium* の独自仕様である状態を伴う関数関数（Section **??**）のための状態変数の検出を行う。この変換では、*mathitdsp* 関数を信号処理のエントリポイントとして，関数が使用する状態変数を，呼び出された状態付き関数名のノードと関数内で使われる状態変数の型を持つ木構造（図中のState Tree）のデータとして出力する。最後に、クロージャ変換された IR と State Tree をもとに LLVM IR を生成する。

ランタイムは、LLVM IR を受け取り、メモリ上でネイティブバイナリにコンパイルする JIT 実行エンジン、オーディオデバイスとの入出力通信を行うオーディオドライバ、時間指定実行された関数とその論理時間の情報を保持するスケジューラの 3 つの部分で構成される。現在、オーディオドライバーには、オーディオデバイスを OS の API で抽象化した C++ 用のクロスプラットフォームライブラリ RtAudio(Scavone 2002) を使用している。実行エンジンは、信号処理のエントリーポイントである *mathitdsp* 関数をオーディオドライバーに

```
ntrigger = 1
fn setN(val:float){
   ntrigger = val
}
fn playN(duration:float)->void{
  setN(1)
  setN(0)@(now+duration)
}
fn Nloop(period:float)->void{
    playN(50)
    nextperiod = if(random()>0) period/2 else period
    Nloop(period)@(now+nextperiod)
}
Nloop(12000)
```

Listing 6.3: Example of Temporal Recursion

渡す。オーディオドライバーは、ハードウェアからの要求を基にスケジューラーに論理時間を進めるように命令し、スケジューラは、論理時間に基づいてタスクを実行したり、実行されているプログラムからタスクの登録の要求への応答、内部時間 (*now*) をプログラムへ返す役割を担う。

LLVM IR まで変換されたコードの中では、タスクの登録と、内部時間の取得という 2 つの機能だけがランタイムに依存する。それ以外のコードは、メモリ上でコンパイルされて実行されるため、C 言語などの低レベル言語で書かれた処理と同等の実行速度を実現できる。

6.5 *mimium* 固有の言語仕様

6.5.1 @ 演算子によるスケジューリング

mimium で時間方向に離散的に発生するイベントを記述するには、Impromptu（Extempore の前身）で導入され、Overtone(Aaron and Blackwell 2013) や Kronos Meta-Sequener(Norilo 2016a) といった言語などでも利用されてきた、継時再帰（**Temporal Recursion**(Sorensen and Gardner 2010)）と呼ばれるデザインパターンを用いる。継時再帰は、一定時間後に関数を呼び出す機能を用いて、ある関数の中で自分自身を一定の遅延とともに再帰的に呼び出すことで、時間とともに繰り返すイベント処理の記述を可能にするものである。

コード 6.3 に具体的な例を挙げる。関数呼び出しに続けて @ 演算子、さらにその後ろに数値型の式を置くと、その関数はすぐには実行されない。代わりに、時間をキーとした優先順位付きタスクキューに登録され、実行コンテキストは次の文に移る。ランタイムは、オーディオドライバのクロックを基にして、各サンプルを処理する前にタスクキューをチェックし、先頭の要素のキーが現在の論理時間に達していれば、それらを先に実行する。時刻は、ランタイム実行を開始を 0 とした絶対時刻（単位は現在のところサンプル）となっているが、キーワード *now* を用いてランタイムから現在の論理的な時間を取得し相対的な時間を記述できる。

コード 6.3 の場合、*mathitntrigger* という変数は、*mathitNloop* という関数が呼ばれるたびに書き換えられる。*mathitNloop* は自分自身を乱数によって不定期な間隔で再帰的に実行し続けるようになっている。この例に対して Extempore で等価なコードをコード 6.4 に示す。Extempore は *callback* という特殊な関数を使って一時的な再帰を行うが、*mimium* は可読性を高めるために中置演算子 *masit@* を導入している。

mimium は ChucK などと同様に論理時間に基づいた同期的なスケジューリング戦略をとる。これは計算プロセスが変数を書き換えたりするだけで言語の中に閉じた IO を介さない処理の場合、サンプル単位での実行順序が正確に保証される。また実装が単純かつ実行コストが比較的少ない、OS のスケジューラに関与しないため移植性が高いなどの利点がある。一方で、標準入出力、MIDI やネットワークなどの信号を受けたり送っ

```
(define ntrigger 1)
(define setN
    (lambda (val)
        (!set ntrigger 1)))
(define playN
    (lambda (duration)
            (setN 1)
            (callback (+ now duration) 'setN 0)))
(define Nloop
  (lambda (period)
        (playN 50)
        (callback (+ (now) (if (random > 0) (/ period 2) period)) 'Nloop period)))
(Nloop 12000)
```

Listing 6.4: Equivalent code to Listing 6.3 in Extempore

たりする IO 処理をスケジューリングして実行した際に、指定した時間が実時間とは必ずしも正確には一致しないというデメリットがある。

Extempore はこれと対照的な、非同期的スケジューリングを行う。これは IO 処理を含めてスケジュールした関数の実行タイミングを実時間に近づけることができるが、実装に各 OS ごとのスケジューラへの対応が必要だったり、実行コストが論理時間ベースに比べると比較的高いというトレードオフが存在している。

mimium の現場の実装は実装の単純さを重視して論理時間ベースのスケジューラを用いているが、これは言語仕様と紐づいたものではなくあくまでランタイム上の実装次第であり、今後の実装によってはスケジューラの戦略をオプションで切り替え可能にもできる。

6.5.2　状態付き関数を利用した信号処理

本セクションではまず信号処理のための UGen を表現するための適切なセマンティクスとデータ構造について検討する。

UGen に仕様されるデータ構造の比較

UGen は、入力データ（もしくはその配列）を受け取り、何らかの処理をして出力するものだ。これは一見すると純粋な関数として表現できるように見えるが、実際には関数と状態変数のペアというデータ構造を使う必要がある。

例えば、入力を加算・乗算するだけなら純粋な関数で十分だが、フィルターや非線形発振器のように、時間 t から $f(t)$ の写像として表現できない処理を表現する時には、UGen は内部状態を持つ。

したがって、汎用言語上で UGen を表現するためには、典型的にはオブジェクトやクロージャなどの関数と内部状態を組み合わせたデータ構造を利用する。ただし、オブジェクトやクロージャでは複数のプロセッサを用いる時に、一度インスタンス化してから実際の処理を呼び出す必要がある。

mimium は信号処理専用のデータ型を作らなくても、UGen を通常の関数のように使うことができる文法を持たせている。

以降では例として 0 から 1 まで一定の割合で増加し、再び 0 に戻るノコギリ波のような UGen：phasor の表現方法を、オブジェクト、クロージャ、さらに Faust、Vult における関数型の表現という異なる体系間で比較し、最終的に *mimium* で UGen を状態を持つ関数として表現するセマンティクスおよびそのコンパイル手順について解説する。

```
class Phasor{
  double out_tmp=0;
  double process(double freq){
    out_tmp = out_tmp+freq/48000;
    if(out_tmp>1){
      out_tmp = 0;
    }
    return out_tmp;
  }
};
//Instantiation
Phasor phasor1;
Phasor phasor2;
double something(){
  //use instantiated objects
  return phasor1.process(phasor2.process(10) + 1000);
}
```

Listing 6.5: The code of Phasor written with object in C++.

■オブジェクト　オブジェクトとは、メンバー変数のセットと、変数を変更したり他のオブジェクトにメッセージを送ったりするメンバー関数（メソッド）のセットを含むデータ構造である。オブジェクトの場合，内部状態はメンバ定数として定義されている. これを利用するには，あらかじめインスタンス化した上で，メインの処理メソッドを呼び出す必要がある。

■クロージャ　クロージャとは、レキシカルスコープを持つ、関数の中で関数を定義可能な言語で利用できる機である。例えば、関数 A の中で複数のローカル変数 $a, b, c...$ を定義して、その変数 a, b, c を自由変数として参照する関数関数 B を返すようなコードを考える。この時関数 A は、関数 B を返す高次関数であり、A を実行することは、オブジェクトにインスタンスを生成することに相当する。

信号処理をクロージャで記述する場合、コンパイル時に変数の寿命を決めることが難しくなるという問題がある。クロージャが使える言語では、ガベージコレクション (GC) を実装して、変数へのメモリの割り当てと解放を動的に行うことが一般的だが、1 秒間に 48000 回もの関数がリアルタイムで実行される DSP のための言語に GC を導入することは困難である (Dannenberg and Bencina 2005)。クロージャを利用できる数少ない言語として、SuperCollider では、リアルタイムシステムで動作可能な複雑な GC をランタイムに実装しており、Extempore では、ユーザーが手動でメモリと変数の寿命の管理することで解決している。言うなれば、クロージャを用いると、ユーザーか開発者のどちらかが大きな実装コストを負担しなければならない。

The following example(Listing 6.6) is a pseudo-code in Java-Script. It is difficult to use this for signal processing practically as JS works with GC, but we used JS to show an example because it is imperative, easy to read, and closure can be used.

■関数型の表現　Faust や Kronos の信号処理の記述では，一時変数の読み書きなしに UGen の代数的な組み合わせ表現を，内部状態を状態変数の代わりに遅延（delay）関数のような最小セットを組み込みの状態付き関数と、信号の 1 サンプル遅延を伴う再帰的な接続表現を用意することで可能にしている。

コード 6.7 の Faust の例では、オブジェクトやクロージャのように，最初にインスタンス化する必要はなく、phasor 用の状態変数は，コンパイル後に個別の関数呼び出しごとに自動的に確保される.

これらの言語ではそれぞれ、Faust ではそれぞれ 0 以上の入出力を持つ UGen（定数は入力 0、出力が 1 つの関数、+ 演算子は入力が 2 つで出力が 1 つの関数、のように)、Kronos では UGen の入出力がリストとしてシンボル化されており、通常の言語のように、記号が特定のメモリアドレス上のデータに対応しているわけ

```
//pseudo-code in javascript
function makeFilter(tmpinit){
  let out_tmp = tmpinit;
  let process = (freq) => {
   out_tmp = out_tmp+freq/48000;//referring free variable out_tmp
    if(out_tmp>1){
      out_tmp = 0;
    }
    return out_tmp;
  }
  return process;
}
//instantiation
let phasor1 = makePhasor(0);
let phasor2 = makePhasor(1);
function something(){
  return phasor1(phasor2(10) + 1000);
}
```

Listing 6.6: The pseudo-code of Phasor written with Closure in Javascript.

```
phasor(freq) = +(freq/4800) ~ out_tmp
    with{
        out_tmp = _ <: select2(>(1),_,0);
 };
// no need to instantiate.
something = phasor(phasor(10)+1000);
```

Listing 6.7: The code of Phasor written with Faust.

```
fun phasor(freq){
    mem out_tmp;//"mem" variable holds its value over times
  out_tmp = out_tmp+freq/48000;
  if(out_tmp>1){
    out_tmp = 0;
  }
  return out_tmp;
}
fun something(input){
// no need to instantiate.
    return phasor(phasor(10)+1000);
}
```

Listing 6.8: The code of Phasor written with Vult.

ではない。そのためこれらの言語に汎用言語と同等の自己拡張性を期待することは難しい。

Vult 言語 (Ruiz 2020) では、関数定義において、通常の変数宣言である *var* ではなく、キーワード *mem* で変数を宣言すると、破壊的変更された値が時系列で保持され、UGen の内部状態を表現できる。この機能により、Faust と同じくあらかじめインスタンス化しておく必要がなく、通常の関数適用として、内部状態を持つ UGen の接続を表現できる。

Faust、Vult ともに、内部状態を持つ関数は、最初にインスタンスを作成することなく仕様できる。その代わり、内部状態の初期化は関数定義時に決定され、インスタンス作成時にコンストラクタを介して初期値を決

```
fn counter(){
    return self+1
}
```

Listing 6.9: The code of sample counter in *mimium*.

```
fn phasor(freq){
  res = self + freq/48000 // assuming the sample rate is 48000Hz
  return if (res > 1) 0 else res
}
fn dsp(input){
    return phasor(440)+phasor(660)
}
```

Listing 6.10: The code of phasor in *mimium*.

定するような操作はできない。

逆に言えば、内部状態の初期値がほとんどの場合 0 または 0 の配列であるという信号処理のドメイン固有知識を利用することで、インスタンス化のための文法をコンパイラ側で省略し通常の関数定義、関数適用と同じように扱える、とも言えるだろう。

mimium での信号処理の記述

こうした前提を基に、*mimium* では、状態付き関数を UGen のように利用できる文法を備えた。これらの関数は，Vult と同じように，通常の関数適用 $f(x)$ のセマンティクスで呼び出せ、その上で Faust のように delay などの限られた組み込み状態付き関数を使い、状態変数管理を意識せずにステートフル関数を書くことができる。さらに、汎用言語の場合と同様に、変数をメモリ上のデータとしてシンボル化する体系は崩していない。

また、関数定義の中でキーワード *self* を使うことで、1 サンプル前の時刻でその関数が返した戻り値を参照することができ、UGen の再帰的な接続を表現することを可能にしている。*self* は，関数定義の中でのみ使用できる予約語で、時刻 0 では 0 で初期化され、関数の前回の戻り値を得ることができます。コード 6.9 に最もシンプルな *self* の使い方として、1 サンプルごとに 1 増加するようなカウンターの関数の例を示した。

self を用いると、これまでオブジェクトやクロージャといった例で見てきた UGen の phasor を *mimium* 上で定義することができる。例をコード 6.10 に示した。この例では、ユーザーは状態変数の宣言の必要も、関数を利用する際のインスタンス化の必要もないことがわかる。Faust で UGen の再帰的接続の表現のための中置演算子 ~ が式の中で何度も利用できるのに比べて、*mimium* では再帰接続の単位は自然と関数定義の単位に分割される。

さらに、データフローや関数型言語に慣れているユーザーのために、パイプライン演算子 $|>$ を用意し、状態付き関数の呼び出しをプロセッサ間の接続として解釈しやすくする構文を持つ。パイプライン演算子は、F#(Microsoft 2020) などの関数型言語で使われており、ネストした関数呼び出し $h(g(f(arg)))$ を $arg|>f|>g|>h$ と書き換えることができる。

コード 6.11 に通常の関数呼び出しとパイプライン演算子の両方を使って正弦波発振器を定義した例を示す。

これに相当するものを，Faust ではコード 6.12、Cycling'74 Max では図 6.2 で示す。Faust の直列接続演算子 (:) の他にも、ChucK 言語の ChucK 演算子 (=>) など、他の言語でも似たような機能を持つものはあるが、*mimium* のパイプライン演算子は、セマンティクスとしては通常の関数呼び出しと等価であるという点が異なっている。

```
fn scaleTwopi(input){
  return input* 2 * 3.141595
}
fn osc(freq){ //normal function call
  return cos(scaleTwipi(phasor(freq)))
}
fn osc(freq){ //pipeline operator version
  return freq |> phasor |> scaleTwopi |> cos
}
```

Listing 6.11: Example of Pipeleine Operator in *mimium*

```
scaleTwopi(input) = input * 2 * 3.141595;
osc(freq) = freq:phasor:scaleTwopi : cos;
```

Listing 6.12: Example of Sequential Composition in Faust

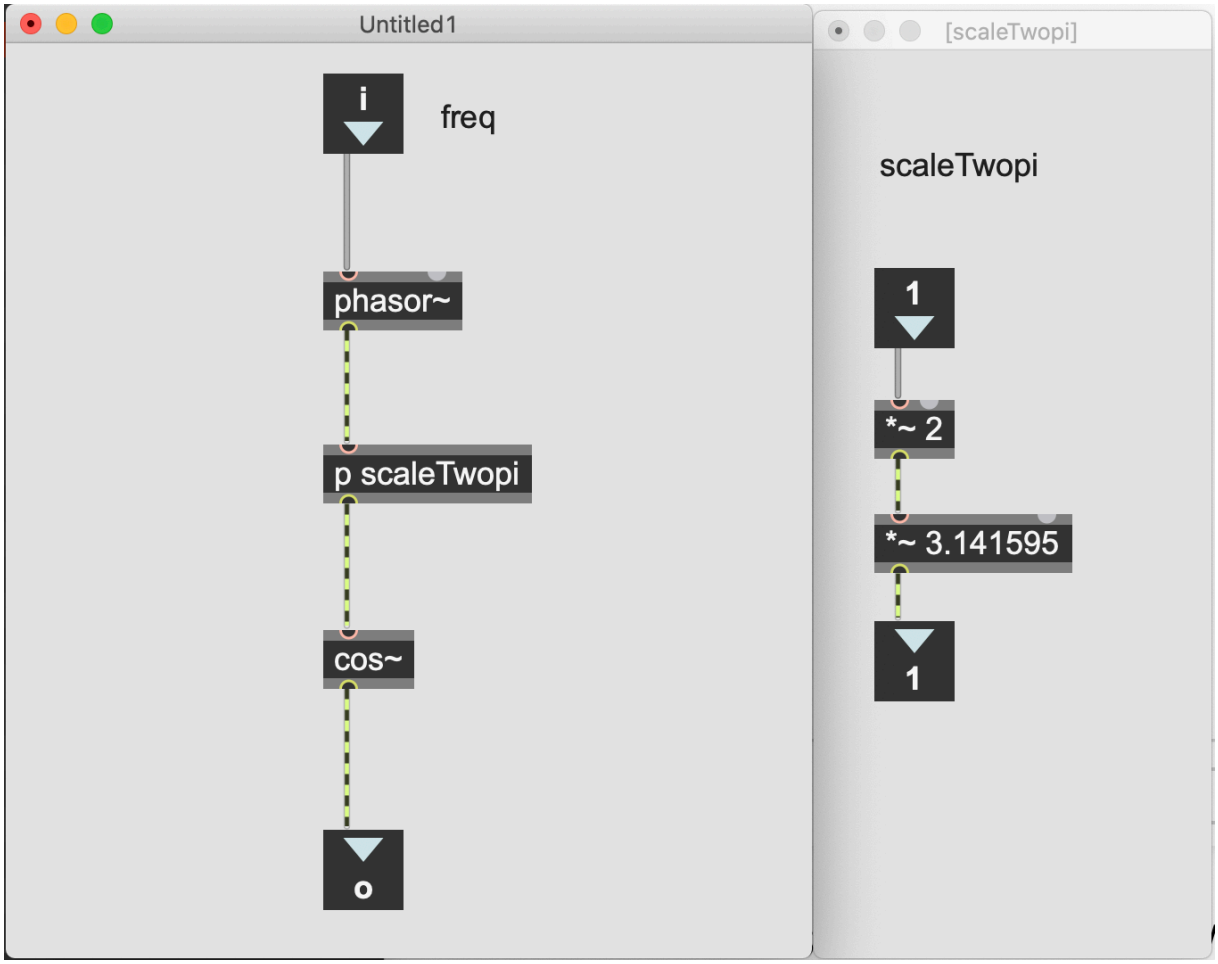


図 6.2: Example of dataflow syntax in Max

状態付き関数のコンパイル手順

状態付き関数は，図 6.1 にあるように、コンパイラにおけるクロージャ変換と低レベルコード生成の間で、状態変数を引数として渡す純粋な関数へと変換される。

```
1  // delay is a built-in stateful function
2  fn fbdelay(input,time,fb){
3      return delay(input+self*fb,time)
4  }
5  fn dsp(){
6      // mix 2 feedback delay with different parameters
7      src = random()*0.1
8      out = fbdelay(src,1000,0.8)+ fbdelay(src,2000,0.5)
9      return (out,out)
10 }
```

Listing 6.13: Example of Feedback Delay in *mimium* before state tree transformation.

```
1 // pseudo-code after lifting stateful function
2 fn fbdelay(state,input,time,fb){
3   //unpack state variables
4     self,delay_mem = state
5     return delay(delay_mem, input+self*fb,time)
6 }
7 fn dsp(state){
8     // unpack state variables
9         s_fbdelay0, s_fbdelay1 = state
10    src = random()*0.1
11    out = fbdelay(s_fbdelay0,src,1000,0.8) +fbdelay(s_fbdelay1,src,2000,0.5)
12    return (out,out)
13 }
```

Listing 6.14: Pseudo-code of Feedback Delay in *mimium* after state tree transformation.

変換は以下のように行われる。まず、*dsp* の関数定義に含まれるすべての関数呼び出しを順に検索し、その関数が *mimium* のコード上で定義されている場合は、さらにその定義を再帰的に検索して、関数呼び出しの依存関係ツリーを作成する。最終的に、その関数定義で *self* を参照する、*delay* などの組み込み状態付き関数を呼び出す場合、その関数は状態付き関数となり、さらにユーザー定義の状態付き関数を呼び出す関数は再帰的に状態付き関数となる。ツリーを作成した後、状態付き関数の引数に、その関数で使用されるすべての状態変数を列挙したタプル型変数へのポインタを付け加えるように関数定義を書き換え、関数呼び出しの部分も同様に、状態変数が明示的な引数になるように書き換える。例として、組み込み状態付き関数である delay と self を利用したフィードバックディレイである *fbdelay* 関数を定義し、*dsp* の中で *fbdelay* を 2 つ、異なるパラメータで呼び出すようなコードをコード 6.13 に示す。

この時の状態変数を明示的に引数として渡すように変換した後の疑似コードをコード 6.14 に示す。

6.5.3 言語仕様の整理、既存の言語との比較

まとめると、*mimium* は、UGen を状態付きな関数として表現でき、時間離散的な制御や信号処理を統一的なセマンティクスで記述でき、ユーザーはハードウェアを意識することなくコードを書くことができる。また、ほとんどのコードが LLVM を介してメモリ上でコンパイルされるため、低レベル言語と同等の実行速度を実現している。時間方向に離散的な処理の記述では、@ 演算子を用いて、関数を実行する時間を指定することができ、継時再帰のデザインパターンと組み合わせることで、繰り返し発生するイベントを抽象化することが可能である。信号処理の記述では、Faust のように状態変数を隠し、*self* を用いたフィードバック接続と組み込み状態付き関数のみを組み合わせることで、内部に状態を持つ関数を通常の関数定義や適用と同じ構文で

表現することができる。

既存の環境と比較して、*mimium* は、**汎用プログラミング言語の仕様と実装に、最小限の音楽のための機能とセマンティクスを付加するアーキテクチャ**をとることで、プログラミング言語の自己拡張性を損なうことなく、実装をシンプルに保ち、ユーザーが音楽的作業に集中できるようにしている。

実際、スケジューリング機能や状態付き関数が使われていない場合、*mimium* は汎用のスクリプト言語のように使える。また、*mimium* のコンパイラの実装は、状態付き関数の変換部分を除けば、一般的な関数型言語のコンパイラと同じである。

Extempore は、単一の環境ですべての記述ができるという点で、このアプローチと似ているものの、ユーザは xtlang と Scheme という 2 つの異なる言語を使用する必要がある。xtlang では、UGen をクロージャとして定義する際に、手動のメモリ管理やポインタを含む複雑な型シグネチャを理解する必要がある。Extempore は音楽に限らないフルスタックのライブコーディングを行うことを想定した環境であるため、手動メモリ管理は必ずしもマイナスポイントではないが、SuperCollider の開発者である McCartney が言うように、音楽のために作られた言語であれば、ユーザーが音楽の作業に集中できるように、メモリやスレッドなどのハードウェア管理を不要、もしくはオプショナルにすることは一般的に重要だと言える (McCartney 2002, p61)。

Kronos(と Meta-Sequencer) も同様に自己拡張性を重視した言語である。Kronos は System $F\omega$ をベースにしたより厳密な関数型言語で、プロセッサの入出力をリストとしてパラメータ化することで、より UGen の一般的な表現が可能である。しかし、その内部表現は Faust のようにグラフ構造をベースにしている (Norilo 2016b, p23)。*mimium* の内部表現は、ラムダ計算ベースの木構造や SSA 形式の命令型 IR であり、より汎用プログラミング言語の IR に近いという違いがある。

第 7 章

議論：音楽土木工学に向けて

7.1 序言

前章では設計した音楽プログラミング言語 mimium の仕様と特徴を例示してきた。

mimium はラムダ計算ベースの汎用プログラミング言語の設計、実装の上に最低限の音楽向けの機能ー論理時間ベースのスケジューラと `delay` や `self` を用いた時間方向の参照による表現をという大きく分けて 2 つの機能を足すことで、汎用プログラミング言語の自己拡張性を失うことなくブラックボックス的構造によるアクセス不可能性を可能な限り排すことを目指した。

本章では前半、言語の実装において未だ不十分である機能の分析を通し、その過程をプログラミング言語に限らない、音楽におけるコンピューターの使い方全般の議論に一般化することを試みる。後半は、2〜5 章でそれぞれ大きな文脈から焦点を絞る形で行ってきた音楽プログラミング言語研究の文脈の構築を、今度は**逆順で**、焦点を広げていく形でそれぞれの章ごとの話題について mimium 設計、実装を通じた貢献を改めて振り返る。つまり音楽プログラミング言語設計における実装の方針と特性のトレードオフという選択の中での mimium の設計の選択（あるいは、非選択)、歴史的な観点でみた音楽インフラストラクチャとしてのプログラミング言語設計の位置付け、デザイン実践を通じた研究 (RtD) としてのプログラミング言語設計の学術的貢献、2020 年代においてコンピューターを積極的に使う音楽家の実践のあり方、という順番になる。

7.2 現状の実装の問題点

まず、現状 mimium を実用的に利用する際に 3 つの問題点が残っており、これらについて解説する。1 つは離散的なイベントの記述は命令型、信号処理は関数型のパラダイムという 2 つの記法のミスマッチ、2 つ目は信号処理に用いる状態付き関数をパラメトリックに複製できないこと、最後が FFT に代表されるサンプルをあるまとまった数ごとに処理をする信号処理が記述できないことだ。

7.2.1 離散イベントの処理と信号処理の記述のミスマッチ

`@`演算子を用いて関数実行を時間指定するという方式は、当然ながら即時その関数が実行されないために、返り値を持たない void 型の関数を用いて副作用（変数の破壊的代入もしくは標準出力への書き込みなどの IO 操作）を起こす、命令型プログラミングのスタイルに帰着する。これは mimium における信号処理で行われる、返り値を次の関数の引数として渡していくことで、信号フローの表現を行うようなスタイルとは大きく異なる。

シンタックス上はコード 7.1 のように、継時再帰とクロージャを組み合わせれば、離散的な値の変化を連続的な信号処理のスタイルと同様に表現することができる。だが、現在の実装ではこのコードは実現不可能である。mimium の現在の実装では関数内部で定義されたローカル変数は本来のスコープを超えて生存することができないため、変数の参照を返り値として返却するようなこのコードを実行することができないためである。

```
fn frp_constructor(period){
    n = 0
    modifier = |x|{
        n = x //capture freevar
    modifier(n+1)@(now+period)
    }
    modifier(0)@0
    get = ||{ n }
    return get
}
val = frp_constructor(1000)
event_val = val()
```

Listing 7.1: Example of encapsulating a temporal discrete value not realized in current implementation of *mimium*.

```
fn filterbank(N,input,lowestfreq, margin,Q,filter){
    if(N>0){
        return filter(input,lowestfreq+N*margin,Q)
                +  filterbank(N-1,input, lowestfreq,margin,Q,filter)
    }else{
        return 0
    }}
```

Listing 7.2: Example of parametric replication of signal processor that cannot be realized in current implementation of *mimium*.

コンパイラが仮にライフタイム解析 (Ruggieri and Murtagh 1988) などの静的解析を用いてコンパイル時に変数の寿命を決定（あるいは、プログラム実行中は永続的に解放されないことを静的に判定）できればこの時間方向に離散的なタイミングで変化する数値の抽象化が可能になる。

7.2.2　状態付き変数のパラメトリックな複製

もう 1 つの問題は、現在の実装では `self` や `delay` を用いる状態付きの関数をパラメトリックに複製できないということだ。コード 7.2 のような例を考えてみよう。関数 `filterbank` は引数 `N` の条件を基にして、別の引数として与えられる `filter` 関数を再帰的に足し合わせるようなコードだ。この時、`filter` 関数に用いられる状態変数の数と、そのためのメモリのサイズはコンパイル時に静的に決定されねばならない。しかし filter 関数が何回複製されるかは引数である `N` によって決定されるために、`filterbank` 関数が実行されるタイミングになるまで状態変数の数の決定は不可能である。

このようなパラメトリックな信号処理プロセッサの複製は、たとえば Faust ではパターンマッチを用いたコンパイル時再帰によって記述可能である。これはオリジナルの Faust のセマンティクスの中には存在しなかった機能で、Gräf による項書き換えシステム—パターンに応じてコードそのものをコンパイル前に書き換えてしまう一種のマクロの導入によって解決されたものである (Gräf 2010)。それ故、コンパイル時にコード 7.2 でいう `N` に相当する、コンパイル時に展開される変数が、整数の定数であることを保証できない場合にはコンパイルエラーになる。加えて、Faust の Block-diagram Algebra の意味論とマクロの体系は別物として存在しているため、コードを書いている人からすると、コンパイル時定数と実行時に変化する変数はどちらも同じ数値であり区別がつけづらい。

mimium においてこの問題を解決するためには、コンパイラが定数を予め展開してしまうような仕組み、あ

るいはマクロを最低でも状態変数の解析の前のステップに導入する必要がある。これはコンパイラが暗黙的に行うことでも解決は可能だが、Faust 同様、ユーザーからすれば `N` が定数でなければいけないことは意味論の外側の事情になってしまうので理解が複雑になってしまう。

この問題は一歩引いて考えると、そもそもこの `filterbank` 関数は、シグナルフローをコンパイル時に決定する処理の記述と、実際にプログラムが実行されるタイミングで発生する信号処理という、2 つの時間における計算を混在させていることになる。こうした状況は MetaML に代表される一種の多段階計算 (Taha and Sheard 1997) と類似した状況と考えられる。多段階計算は、計算が起きるステップを明示的に文法の中に記述でき、型システムにもその変数がいつ計算される変数かという情報を入れることができるような計算パラダイムで、再帰の発生する計算などを効率的にプログラムできるといった特徴がある。

計算がいつ発生するかという視点で考えれば音楽プログラミング言語には、シグナルフローの決定という計算以外にも例えば波形テーブルの生成、リズムパターンの生成といった、プログラムの中では一度きりしか発生しない計算はいくつもあるため、導入することで副次的に得られる利点もあると考えられる。

さらに、MacroML(Ganz, Sabry, and Taha 2001) のように、多段階計算を安全なマクロとして考えると、言語の自己反映性を高め、言語内 DSL をライブラリとして実装できるような応用方法も考えられる。

例えば、Puckette はサンプリング理論に対する不満、と名付けた論考の中で、連続領域で（例えば微分方程式などの形で）構築した楽器や電気回路の数学的モデルを離散化してから計算するような例を挙げ、すべての表現がサンプリング理論に基づく、各時刻における音圧に対応した数値を計算するような音楽表現には一定の限界があることを指摘している (Puckette 2015)。たしかに、プログラミング言語を使うということは、シンボルを組み合わせることである（ときには実在しないかもしれない）現象のモデルを記述し、それをコンピューターに実行させているということなのだから、例えばバネ–マス–ダンパといったシンボルを組み合わせてある力学系のモデルをテキストとして記述することは、モデルが連続領域であろうと離散領域であろうと可能だし、連続領域のモデルとして記述されたシンボルの組み合わせを離散化する関数に通し、信号処理に利用する、といった方法の記述は 1 つのソースコードの中に収めることも可能なはずだ (実際、Wolfram のような数値計算のための言語ではまさにそのような記述が可能である[*1])。これもやはり、モデルを離散化するという計算が大抵はコンパイルしたタイミングで発生しており、いつ計算するかの意味論が存在しないことにより、連続領域のモデル記述は一度離散化したデータを読み込むといったプロセスを経なければ記述ができないという見方ができるだろう。この考え方をもう少し進めれば、例えば微分方程式のモデルを記述する、という作業を機械学習におけるモデル学習のための関数を記述する、という置き換えもできる。もちろん、現実的には機械学習のようなモデル学習に計算リソースを激しく使うようなケースでは、コードを実行するたびに学習し直しということになっては非効率極まりないので、計算した結果をキャッシュするような仕組みがコンパイラに必要になるだろうが。

言い方を変えると、Wav ファイルによるサンプルやウェーブテーブルも、連続領域モデルを離散化した後の重みづけパラメーターのような計算結果も、機械学習における学習語のモデルデータも、**何かしらのモデルから生成されたキャッシュ**のようなものと捉えることもできるだろう。この、いつ計算するかを意味論に明示的に加えていくことは、現在の音楽情報処理において解離してしまっている、python を用いて学習、Max などのツールを用いてモデルを実行し音声をリアルタイムで出力といったような 2 つの作業を 1 つながりの作業としてつなぎ合わせるような役割を音楽プログラミング言語に与えることに繋がる。

7.2.3 複数サンプルレートが混在した信号処理

mimium の信号処理は Faust などと同様、1 サンプルごとの処理の記述によって表現されているため、FFT に代表されるような、256、512 などまとまった数のサンプルを処理し、同じまとまりのサンプルを一挙に

[*1] https://reference.wolfram.com/language/howto/SolveADifferentialEquation.html.ja?source=footer

```
fn counter(incl:float){
    return self+incl
}
```

Listing 7.3: 1 サンプルずつ増加するカウンター

出力する、いわゆる複数サンプルレートの混在した信号処理（Multi-rate Processing）の記述が不可能である。Kronos はサンプルレート自体を変更するコンパイラ組み込みの関数を導入することでこれを解決している (Norilo 2015, p40˜41)。Faust でも Multi-rate 処理の実現のための文法拡張は同様に試みられている (Jouvelot and Orlarey 2011) が、これも多段階計算同様に**いつ計算するか**に関わる、一般的なプログラミング言語では考慮されない要素であるため、どのみち意味論に何かしらの拡張を加えなければ根本的な解決はできないものと考えられる。

7.3　より形式的な定義のための関連研究

現在のところ、mimium のシンタックスの定義に関しては付録で示すように形式的な定義を行えてはいるが、意味論、特に、`self` のような状態付き関数の扱いは実装によって示されているのみである。この参考となる研究が mimium を発表した ACM SIGPLAN FARM(Functional Art, Research, Modeling) というワークショップで同じセッションで発表された W 計算（W-calculus、ラムダ計算と同じような名付け）である。W 計算は Faust に強く影響を受けていながらも、Faust のように入出力を持ったブロックのグラフを組み合わせる構造ではなく、ラムダ計算をベースにした信号処理のための計算で、通常のラムダ計算に現れる項に加えて、$feedx.e$ といった独自の項が追加されている。この `feed` はその項における 1 時刻前のサンプルでの計算結果が x として項の中で再帰的に利用できるといった意味合いをもち、これは mimium における `self` の概念に直接的に対応する。例えば 1 サンプルずつ引数 `incl` に応じて増加するカウンターの関数コード 7.3 は W 計算の中では次のように表せる。

$$\lambda\, incl.feed\, self.(incl + self)$$

W 計算はディレイのような、過去のサンプルを参照するような意味合いを定義すると、実用的には過去のサンプル全てを保存しなければならなくなってしまうといった、形式化に伴う問題に対しても都度意味論を拡張することで対応できている。また、OCaml で書かれた W 計算で記述された信号処理のプログラムを、W 計算のインタプリタとともに一度 MetaOCaml 上で評価することで、入力プログラムに特化したインタプリタプログラムを生成し、コンパイラの最適化をさらに組み合わせることでリアルタイムでの計算も実現できるようになっている、Multi-rate 信号処理への拡張も既に方針が示されているなど、実用を意識した上での理論構築がなされている。

mimium も根本的な意味論の定義を W 計算の形式を借りることによって可能になるだろうと考えられる。もっとも、W 計算においても時間方向に離散的な処理の意味論は含まれておらず、現状のランタイムとの通信という実装に依存する定義方法に対する解は今後別に検討する必要がある。

7.4　実装の方針とトレードオフの選択

第 5 章で整理した、音楽プログラミング言語の実装の方針と、それに伴う特性のトレードオフが mimium ではどうなっていたかについて改めて整理しよう。

まず、実装の方針としては、自己反映性が高いホスト言語に乗る形での Internal DSL とするか、文字列データの解析から行う External DSL とするか、あるいはそのハイブリッドの形にするかという視点があった。こ

こについては mimium は完全な External DSL となっている。コンパイラ・コンパイラである bison を用いることで文字列解析の実装そのものは SuperCollider や Faust 同様に下げられているが、C++ という低レベルプログラミング言語での実装を行っているためその実装の手間（開発コスト）は少なくない。

よりテクニカルな問題として C++ を基盤としない言語を用いて開発し直すという選択肢も考えられる。具体的には、C++（17）を用いての実装において、実装の参考にした OCaml と比べると代数データ型の扱いが複雑になることが大きな障害となった。C++17 においては、標準ライブラリに `std::variant` という、代数データ型における直和型（N 種類の型のうちどれか 1 種類が含まれているような状態を示す型）が利用できるようになったが、これは Abstract Syntax Tree を表現するための木構造に用いる再帰する代数データ型 (新しく型を定義するときに自分自身の項が含まれるような型）の定義を少し遠回りな方法でなければ実現できないことによる[*2]。加えて、N 個の variant の中から各データ型に対応する処理を記述する際、OCaml 等の関数型言語においてはパターンマッチと呼ばれる記法を用いて短く処理を記述することができる。C++ で variant を使った場合、テンプレートによる静的ポリモーフィズムの複雑な記述を行う必要がありこれも開発の効率を落とす一つの要因であった。そもそも C++ を開発言語として選択したのは、第 5 章で説明した通り音楽のための言語を実装するにあたって、メモリの確保を明示的に制御できるような言語である必要性があることを意識していたという理由があるが、これは今にして思えばランタイムの実装（C++ で書かれたプログラムならコンパイルされそのプログラムは静的にメモリを確保して動く）とコンパイラの実装 (何かしらの言語で書かれたプログラムが LLVM ライブラリなどを通じて静的にメモリを確保するようなプログラムを動的に出力する) を混同していたため、コンパイラ部分だけの実装を別の言語で実装することは可能である。ただ、幅広いプラットフォームをサポートしようと思えば LLVM が利用できることは重要な要素であるため、公式でそれらがサポートされている OCaml や、C 言語 API をラップしたライブラリが存在している Haskell や Rust のような言語が異なる実装のための候補として挙げられる。

ともあれ、mimium における独自に新しくシンタックスの定義を行っているため、（その表面上の見た目は Rust という既存のポピュラーな言語に寄せてあるものの）Internal DSL として実装されている言語と比べてるとその学習コストは高いし、ホスト言語の既存のライブラリ資源の活用も難しい。つまり、既存の音楽プログラミング言語の利用者からの参入障壁が大きい。これはインフラストラクチャとしての言語という側面から考えると重要な問題で、給水塔という既存のインフラストラクチャに乗ることで広がる携帯電話の基地局という新しいインフラストラクチャ (Parks and Starosielski 2015, p2) のように、新しいインフラストラクチャを整備するために既存のインフラストラクチャを利用することは必須とは言わずとも、利用しないことによってハードルが格段に上がってしまう。設計当初、言語自体の表現力が高くできてさえいればあまり問題にならないだろうと考えていたが現在はその考えは間違えだった。すでに繰り返し述べているように音楽プログラミング言語の実装には時間がかかる。言語自体の実装やアップデートもそうだし、言語上で構築するライブラリはなおのことである。なぜなら、言語上で開発されたライブラリは言語の中核的な機能や、include のようなモジュール読み込みのための機能が更新されたり変更されるのに伴って大きく変更される必要がままあるため、本腰を入れて開発を始めることが難しいからである。そうしたときに、外部の言語のライブラリを活用できるということは、言語のコアな機能の更新をチェックしながらも実用性を失わずに存在し続けることができるということになる。

今後このような言語間の相互運用性を高めるための方法としては 2 つの方針が考えられる。1 つは、別の言語の関数呼び出しの中で最も実装が簡単な C 言語ライブラリの関数呼び出しを、mimium 上でライブラリ名、関数名、型名を書けば実行時にリンクして呼び出せるようにする、いわゆる Foreign Function Interface(FFI) の機能を追加することだ。ただ、簡単とはいえ、mimium では意味論上隠蔽されているハードウェア的な要素が含まれる型の扱い、つまりポインタ型の扱いを型システムのシンタックスにも追加しなければならないため、簡単とはいえコンパイラに入れる手間は比較的複雑なものとなる。2 つ目は、mimium 自体を Faust のように、

[*2] `std::variant` の実装の元となった boost ライブラリの中には再帰する variant が記述可能なものもある。URL などで補足する

Max や SuperCollider の UGen として扱えるようなワークフローを整備することで他の言語の拡張機能的に使えるようにするという方法だ。これは既に、mimium 自体のアーキテクチャをなるべく疎結合にしていたこともあり、アーキテクチャの図におけるオーディオドライバの部分に当たる箇所を、Max や SuperCollider それぞれの UGen 作成のための API を用いて実装すればよいため、C 言語 FFI の実装と比べると比較的簡単ではある。

(トレードオフの選択に関してもう少し詳細な記述をしたい)

7.5　歴史的観点

音楽プログラミング言語は歴史的には、80 年代以前はあらゆる種類の音楽制作ソフトウェアの起源としての意味合いが強く、90 年代の SuperCollider や Max、Puredata のような、音楽のための最低限の抽象化を施すことで、音楽プログラミング言語を学ぶためにまずそれより低レベルな言語について学ぶような状況を避けつつ、表現は既存の音楽様式に縛られないような自由度を持つといったバランスをとった言語によってその意義が決定づけられたという流れがあった。さらに 2000 年代は、既存の言語では組み込みのもの（= UGen）として与えられていたものまでもプログラマブルにできるような、Faust に代表される低レイヤーの拡張、言語の意味論自体に自己反映性の限界があることに対して、既存の汎用言語の表現力を借りることによって突破するような高レイヤーの拡張、オペレーティングシステムに依存してしまう荒いタイミング制御をさらに正確にするような取り組みという 3 つの方向性が現れてきていた。

この中で mimium は主に低レイヤーの拡張に重きを置きつつ、既存の言語では信号処理という、時間方向に連続した処理だけでなく、小規模なスケジューラを載せることにより離散的なイベントの実行も射程に入れている。既存の音楽の様式をなるべく言語仕様に埋め込まないようにするべく、汎用言語の設計に最低限の時間制御の言語仕様：関数の時間指定実行と、`self` や `delay` による信号処理における時間方向の参照による表現という 2 つの機能を載せつつ、かつメモリやスレッドのようなハードウェアを意識する必要はない（特定のハードウェアのアーキテクチャにロックインされない）ようなシンタックスという McCartney のいう “音楽のための最低限の抽象化” を現代の状況において可能な範囲で再検討したものだった。

一方であえて重視しなかった機能として、近年の高レイヤーの拡張を試みる言語が同時に行ってきたコードの動的変更、つまりライブコーディングのための工夫はほとんどされていない。とはいえ、全体のアーキテクチャの参考とした Extempore は低レイヤーの表現の自由度を担保しながらライブコーディングができることを目指した言語であり、LLVM という低レイヤーの処理をメモリ上でコンパイルして即時動作させるインフラストラクチャを用いているという共通点を見れば、少し手を加えることで、動的にコードを変更していった時に音声がブツ切れにならないかといった細かい考慮を度外視すれば実現自体は可能だと考えられる。ただし、その機能は LLVM の JIT コンパイルのためのライブラリに依存することになるため、LLVM を使わないバックエンド–具体的には、Web ブラウザ向けの Webassembly バックエンドなどを追加しようと思った時の移植性が低くなってしまうことが予想できる。現在はコードの動的変更のしやすさよりも稼働できるプラットフォームの幅の広さの方に重きを置くべきと考えている。

（この辺ももうちょっと充実させたい）

7.6　音楽プログラミング言語とは何か？

ここまでの視点で一度改めて本研究の提出する貢献、音楽プログラミング言語の存在論についての議論を固めよう。

音楽プログラミング言語とは本質的には McCartney の言ったように、音楽のための最低限の抽象化機構を備えた人工言語である。その最低限をどこに設定するかによって、大きければ Unit Generator のインタプリタのモデルのようになるし、小さければ Faust のような形式的意味論を備える言語体系になる。

プログラミング言語とは現代においてはハードウェアの違いに依存しない抽象的な計算モデルを、人間が記述可能な形で操作できるようにしたものといった意味合いが強い。その中でも、比較的現代的なものを含めて汎用プログラミング言語の下敷きになっている理論は未だ可能な限り速く計算できることが是とされる時間を考慮しない計算モデルだ。また時間を織り交ぜた抽象計算モデルは、少なくともリアルタイムコンピューティング (線形時相論理)、並行計算 (CSP、PI-Calculus)、論理時間に基づいた計算 (Faust/W-calculus) 全てが音楽の処理には現実的には必要でありながら、未だ確立したモデルが存在していない。

そのため、音楽プログラミング言語を汎用プログラミング言語の上にライブラリとして構築しようとしたとき、必要な時間の制御に関する機能や記述方法を、純粋に上のレイヤーに積み重ねて設計するようなことが難しくなる。それ故ホスト言語が本来扱うよりも低レイヤにある、実際のハードウェアに近い操作の記述を行う必要が出てくるという、純粋な抽象化のレイヤー構造が崩れる抽象化の逆転現象が発生する。

そのため、なるべく自由度の高い音楽の記述を行うにはライブラリではなく言語の設計そのものを行う必要があり、それは未だ確たるものが存在しない時間を考慮した抽象計算モデルの中で暫定的なものと、ハードウェアや OS と言ったシステム上で動くランタイムプログラムをどうにか連携させることで動かすというブリコラージュ的なプログラミング言語であるという表現ができるだろう。

7.7 デザイン実践を通じた研究（RtD）として

さて、もう少し視点を広げて本研究がデザイン実践を通じた研究（RtD）としてどのような位置付けができるのかについて検討しよう。

第 3 章前半で論じたように、プログラミング言語を設計する研究はその評価の基準の曖昧さの一方で、各技術要素については定量/定質様々な方法で客観性を持った評価をすることは可能だという HCI におけるインターフェースデザインの主要な研究とはやや趣が異なる研究である。mimium の設計も、第 6 章で論じてきた中心的な機能以外の部分に関しては実用性を考えた別個の機能が飛び飛びに開発されている。例えば本論の中では触れなかった、外部のファイルを読み込む include の機能などは音楽的に関わる意味論の設計の外側であったため、単純な文字列置き換えによる場当たり的実装となっている。一方でこの機能は開発されてなければ様々なコードのテストのファイルはライブラリのような機能分割をしない全て自己完結したコードでなければならない。

このような状態で、本論文の主要な主張 (Claim) たる、音楽において自分のソフトウェアを自分で開発できるようなインフラストラクチャの形成という目標に対して、それを支える根拠（Evidence）は長期的目線での音楽情報インフラストラクチャの形成の必要性とそれに応じたなるべくブラックボックスを少なくするような言語の設計と実装という、いわば設計思想のみであって、個々の技術的要素、例えば、信号処理を LLVM を用いることでコードをメモリ上でコンパイルでき、リアルタイムで高速な処理が行える、といった要素は、例え厳密なベンチマークを行っていたとしても主張を支える十分な根拠とは現状なり得ない。mimium という人工物に対する Annotation としてのこの主張と根拠の結びつきは、今後言語が少しづつ実用される中で初めて評価されることになる。

(やっぱり言い訳がましさが残る気もするので、もうちょっと言い回しを賢くしたい)

7.8 テクノロジーを用いる音楽実践として

最後に、プログラミング言語設計自体を 2020 年代における 1 つの音楽的実践としてできるかについて検討しよう。

アマチュアリズム的態度に準じたハッキング/ティンカリング的態度に基づくテクノロジーの誤用はもはや、アクセス不可能なブラックボックスにより無効化され、コンピューターという象徴機械を介して音楽文化は無意識に画一的な方向に収束する傾向にある。

そうした時代に音楽家が取れる態度はまず、テクノロジーの中身そのものを根本的に理解できるようになっていくよう変わる必要がある。Puckette は Max や Puredata が比較的普及した後でも、それを使うためには教育の要素が不可欠だと主張しているが、この状況は現在になっても変わっていないと言える。

> しかし、コンピュータハードウェアの低廉化によって交わされた約束に、私たちはまだ追いついていない。確かに、いまやすべての道具が入ったコンピュータを 400 ドル程度で購入でき、アンプとスピーカを追加すればコンピュータ音楽制作の準備が整う。しかしこれは、システムを構築し、フリーの良いソフト見つけてインストールし実行するために必要な知識があることを前提としている。コンピュータを数千ドルに、ソフトウェアを数千ドルにしたいという多くの商業的な関心が、私たちの前に立ちはだかる。コンピューティングとコンピュータ音楽の民主化に不可欠な要素は、地域の知識ベースを育成することだ。将来を見据えた音楽教育者は、学生が自分のコンピュータを構築するのを促すために何時間も割いている。自家製コンピュータと自家製コンピュータ音楽ソフトウェアの国際的な文化を、いつか見たいと私は思う。過去に裕福な西側はソフトウェアを開発し、何百万もの CD を作り、買う人には誰にでも販売した。将来的には、他の世界から輸入されたソフトウェアを研究し学習するセンターを見てみたい。知識を育てるコミュニティが、特に Linux のような非商用 OS では必要である。もし Linux を使う友人がいなければ、動かすまでには障害があるだろう。しかし、世界の多くで少なくとも村の 1 人や 2 人に、コンピュータ音楽の専門知識（マシンの組み立て方法、OS のインストール方法、ソフトウェアの実行方法など）がある未来を想像できる。

さらに、テクノロジーを深いところまで理解した上で取れる態度は 2 つに分かれる。ひとつはわかった上で誤用することで、作られたテクノロジーの価値を転換する (**Tokui2021**) という、Huutamo のいう T(h)inkerer 的態度だ。一方でこの態度には、音楽家という人間の外部にテクノロジーを置き、誰かが作ったテクノロジーの変革のタイミングに依存して音楽文化が変化していく、Theberge の MultiSectorial Innovation 的状況を受け入れた上で、即興的に反応して技術の異なる可能性を追求するということにもなる。これは近視眼的な意味でのテクノロジーとそれが生み出す新たな象徴による文化を形成するかもしれないが、Attali の反復の系から抜け出せるほどの新しい音楽文化を作るには一歩足らない。

そうでないやり方として、ここ数年の人工知能の過熱のような論争中のテクノロジーではなく、論争後、例えば MIDI のようにすでに所与のものと決定づけられてしまい、透明な背景と化してしまったインフラストラクチャのような対象を、長い時間がかかるとしても自らの手で再発明し、あり得るかもしれない今（Alternative Present）を生み出すことが必要だ。そしてそれはデモンストレーションとして議論を巻き起こすところまで完結するのではなく、少しづつでも今作り始めて今使い始めてしまうという、技術の可能性の道の間に存在するフィールドにけもの道を作り出すようなアプローチである。

この態度の 1 つのインスタンスが、プログラミング言語開発という手段である。

7.9　音楽プログラミング言語を作るとはどういうことか

RtD としての実践、音楽実践としてのプログラミング言語制作という話題のまとめとして、音楽プログラミング言語を作るとはどういう行為なのか、についてまとめよう。

音楽プログラミング言語とを作るとはすなわち音楽のための最小限の抽象化機構を実装するということだ。これは大きなブラックボックスを持つ UGen パラダイムの言語であれば、オシレーターやフィルターのようなプリミティブ UGen を実装する作業も言語設計、実装の一部となる。しかしブラックボックスを小さくした言語ではオシレーターやフィルターの実装はその言語上で行うライブラリの実装作業となる。

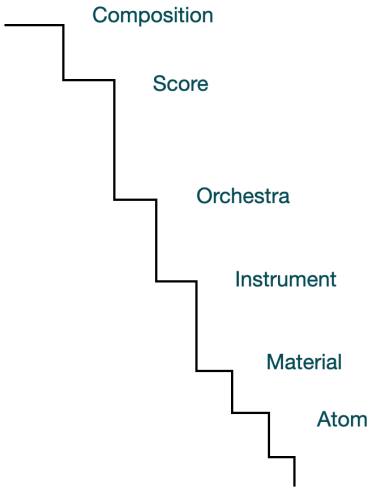


図 7.1: 従来の Multi-Language パラダイムに基づく言語におけるプログラミングの学習経験の差を ”いびつな階段” として概念的に表した図。

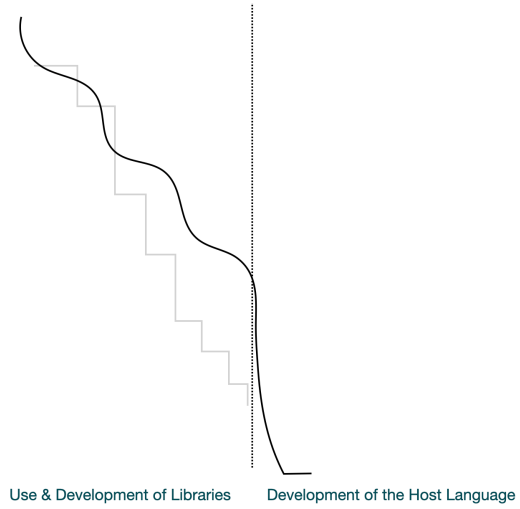


図 7.2: mimium のようなブラックボックスを減らし、ライブラリとしての実装の役割を広げた言語における学習経験の差を ”突然急になる坂” として概念的に表した図。

こうしてなるべく音楽プログラミングにおける作業における経験の階段のギャップを減らし、パパートのいう天井は高く、床は低くといった目標を追求していくと、しかし最終的には肝心のホスト言語の実装そのものはほとんどテキストデータを読み込んでより抽象的なデータ構造へと変換していくデータ変換器と言った様相を呈してきて、階段のギャップはほとんど均一になったが最後の一段だけが飛び抜けて高い、といった切断を引き起こしてしまう矛盾が根本的に存在する。この現象を説明した図が図 7.1 と図 7.2 である。図 7.1 が各ドメインごとに使う言語を分ける、Multi-Language パラダイムに基づく言語の使用と開発、それぞれの経験の差が発生していることを概念的に表したものだ。なお Score-Orchestra-Instrument は CSound で用いられた用語だが、Material や Atom はあくまでメタファーとして独自に用いている。言うなれば Instrument という UGen を C 言語などでプログラミングする作業が Material レベルのプログラム、さらに C 言語のコンパイラそのものを作るような作業が Atom レベルのプログラムとでも言えよう。一方、図 7.2 が mimium のようなブラックボックスを減らし、ライブラリとしての実装の役割を広げた言語に対応する。Papert のいう Floor も Ceiling も領域は広がっているし、各言語間に存在した差異もなだらかになっている一方、言語を作ることそのものの難易度は C 言語などよりも理論的背景を多く必要とし難易度が高くなってしまう。

つまり、音楽プログラミング言語を作るという作業は最終的に “音楽のためのライブラリを作りやすい汎用言語を作る” と “その言語上でライブラリを作る” という 2 つの行為に分裂していく。

このトレードオフは根本的には解決しようがないが、多数ある言語同士の相互運用のしやすさを上げることである程度は解決できるものと思われる（もうちょっと補足）。

7.10　音楽土木工学という学問領域

強いコンピューター音楽： 作品のユニークネスにコンピューターを使用することが関わってくる弱いコンピューター音楽： 頑張ればコンピューターがなくても実現できるけど、コンピューターを使用することで製作/配布/再生/上演が円滑になる

やや弱いコンピューター音楽: 作品を成立させるのにコンピューターが不可欠だが、コンピューターを使っていることは作品のユニークネスとは特に関係がない

トフラーのプロシューマー、CGM、End User Programming

第 8 章

結論

8.1 音楽プログラミング言語とは何か

音楽プログラミング言語とは本質的には McCartney の言ったように、音楽のための最低限の抽象化機構を備えた人工言語である。その最低限をどこに設定するかによって、大きければ Unit Generator のインタプリタのモデルのようになるし、小さければ Faust のような形式的意味論を備える言語体系になる。

プログラミング言語とは現代においてはハードウェアの違いに依存しない抽象的な計算モデルを、人間が記述可能な形で操作できるようにしたものといった意味合いが強い。その中でも、比較的現代的なものを含めて汎用プログラミング言語の下敷きになっている理論は未だ可能な限り速く計算できることが是とされる時間を考慮しない計算モデルだ。また時間を織り交ぜた抽象計算モデルは、少なくともリアルタイムコンピューティング (線形時相論理)、並行計算 (CSP、PI-Calculus)、論理時間に基づいた計算 (Faust/W-calculus) 全てが音楽の処理には現実的には必要でありながら、未だ確立したモデルが存在していない。

そのため、音楽プログラミング言語を汎用プログラミング言語の上にライブラリとして構築しようとしたとき、必要な時間の制御に関する機能や記述方法を、純粋に上のレイヤーに積み重ねて設計するようなことが難しくなる。それ故ホスト言語が本来扱うよりも低レイヤにある、実際のハードウェアに近い操作の記述を行う必要が出てくるという、純粋な抽象化のレイヤー構造が崩れる抽象化の逆転現象が発生する。

そのため、なるべく自由度の高い音楽の記述を行うにはライブラリではなく言語の設計そのものを行う必要があり、それは未だ確たるものが存在しない時間を考慮した抽象計算モデルの中で暫定的なものと、ハードウェアや OS と言ったシステム上で動くランタイムプログラムをどうにか連携させることで動かすというブリコラージュ的なプログラミング言語であるという表現ができるだろう。

その中でも mimium という言語は˜

8.2 音楽プログラミング言語を作るとはどういうことか

音楽プログラミング言語とは上記のような性質を持っているため、それを作るとはすなわち最小限の抽象化機構を実装するということだ。これは大きなブラックボックスを持つ UGen パラダイムの言語であれば、オシレーターやフィルターのようなプリミティブ UGen を実装する作業も言語設計、実装の一部となる。しかしブラックボックスを小さくした言語ではオシレーターやフィルターの実装はその言語上で行うライブラリの実装作業となる。こうしてなるべく音楽プログラミングにおける作業における経験の階段のギャップを減らし、パパートのいう天井は高く、床は低くといった目標を追求していくと、しかし最終的には肝心のホスト言語の実装そのものはほとんどテキストデータを読み込んでより抽象的なデータ構造へと変換していくデータ変換器と言った様相を呈してきて、階段のギャップはほとんど均一になったが最後の一段だけが飛び抜けて高い、といった切断を引き起こしてしまう矛盾がある。

8.3　音楽プログラミング言語を研究するとはどういうことか

これにはいくつかの暫定的かつプラグマティックな回答が考えられよう。

1 つは、多段階計算や W-Calculus のような時間を含めた理論的な計算モデルの研究が音楽以外の役に立つであろうことを示すことで、音楽表現のためのライブラリ作りと、その基礎となる抽象計算モデルの構築との分離が引き起こされても、後者を研究する人が常に一定数存在してくれるので問題がないだろうとする立場だ。

もう 1 つの立場は、そもそも計算モデルをハードウェアから完全に抽象的に分離することなど不可能だとする立場だ。この場合、今後より進むだろうと考えられるドメイン固有アーキテクチャのような、特定の目的に特化したデバイスを作るためにコンピューターアーキテクチャやその上に乗るソフトウェアインフラストラクチャ自体もそれに合わせて柔軟に設計するという方針を取る。この時例えば、フィルターやオシレーターといった比較的高次に抽象化されたシンボルは、例えば FPGA のようなデジタル回路だがプログラム内蔵方式の計算とは違いハードウェア的に信号のフローを決定できるようなものから、さらには直接アナログ回路のハードウェアへ変換し製造されるといった例も考えられる。その時音楽プログラミング言語に必要なのは、任意の抽象度の関数や処理内容を任意の抽象度のまま多様な形式のハードウェアのレイヤーへと変換できるようなユーティリティツールだろう。これは汎用プログラミング言語における LLVM や、それをさらに高次のレイヤーへ拡張した MLIR のようなプロジェクトに対応する、やや泥臭い作業が必要なプロジェクトになるだろう。

今後の mimium の開発方針およびそれを通じた研究は、おそらくはこの 2 つのアプローチを同時に進めるものになるだろうと考えている。

論理時間に基づいた信号処理と、論理時間に基づかなくともよい離散的なイベント処理の二つは計算モデルとしては統合されておらず、後者をランタイムのサポートでどうにかする状態になっている。

デザインが取り扱う領域は日々広がっており、その領域は法や都市、自然環境、経済といった抽象的なもの—マルクスの言う上部構造（≒より概念的なもの）へと及んでいっている。しかしコンピューターという記号を処理する機械に囲まれて生きる私たちには、同時にその機械のやり取りの根底に用いられているプロトコルやフォーマットといった下部構造にもまた大きく影響を受けており、そのプロトコルやフォーマットはまた上部構造をエンコードしたものでもある。時間のかかる意地悪な問題を取り扱うデザインの実践—例えばトランジションデザインはその最上部、最下部の接点に目を向けることで変化のためのヒントを見出すことができるだろう。

現状答えが出ておらず、Open Question として残されているのは、このような、変化に時間がかかる研究にどのようにインセンティブを持たせていけばいいのかということである。

参考文献

Aaron, Samuel and Alan F. Blackwell (2013). “From Sonic Pi to overtone: Creative musical experiences with domain-specific and functional languages”. In: Proceedings of the ACM SIGPLAN International Conference on Fu pp. 35–46. DOI: 10.1145/2505341.2505346 (cit. on p. 69).

Anderson, David P. and Ron Kuivila (1990). “A System for Computer Music Performance”. In: ACM Transactions on Computer Systems (TOCS) 8.1, pp. 56–82. ISSN: 15577333. DOI: 10.1145/77648.77652 (cit. on p. 59).

Born, Georgina (1995). Rationalizing Culture. 1. University of California Press. ISBN: 0520202163 (cit. on p. 13).

Brandt, Eli (2002). “Temporal type constructors for computer music programming”. PhD Thesis. School of Computer Science, Carnegie Mellon University, p. 168 (cit. on p. 57).

Coblenz, Michael et al. (2018). “Interdisciplinary Programming Language Design”. In: Proceedings of the 2018 ACM SIGP New York, NY, USA: ACM, p. 14. ISBN: 9781450360319. URL: https://doi.org/10.1145/3276954. (cit. on p. 58).

Culpepper, Ryan et al. (2019). “From macros to DSLs: The evolution of racket”. In: Leibniz International Proceedings in In Vol. 136. Schloss Dagstuhl- Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing. ISBN: 9783959771139. DOI: 10.4230/LIPIcs.SNAPL.2019.5 (cit. on p. 55).

Dannenberg, Roger B. (1997). “The Implementation of Nyquist, A Sound Synthesis Language”. In: Computer Music Journal 21.3, pp. 71–82. ISSN: 01489267, 15315169. URL: http://www.jstor.org/stable/3681015 (cit. on p. 42).

– (2018). “Languages for Computer Music”. In: Frontiers in Digital Humanities 5. ISSN: 2297-2668. DOI: 10.3389/fdigh.2018.00026. URL: https://www.frontiersin.org/article/10.3389/fdigh.2018.00026/full (cit. on pp. 42, 50, 58).

Dannenberg, Roger B. and Ross Bencina (2005). “Design Patterns for Real-Time Computer Music Systems”. In: ICMC 2005 Workshop on Real Time Systems Concepts for Computer Music. URL: https://www.researchgate.net/publication/242648768_Design_Patterns_for_Real-Time_Computer_Music_Systems (cit. on p. 71).

Ganz, Steven E., Amr Sabry, and Walid Taha (2001). “Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML”. In: SIGPLAN Notices (ACM Special Interest Group on Programming Langua 36.10, pp. 74–85. ISSN: 03621340. DOI: 10.1145/507669.507646. URL: https://dl-acm-org.anywhere.lib.kyushu-u.ac.jp/doi/abs/10.1145/507669.507646 (cit. on p. 79).

Gräf, Albert (2010). “Term Rewriting Extension for the Faust Programming Language”. In: International Linux Audio Conference. URL: https://hal.archives-ouvertes.fr/hal-03162973%20https://hal.archives-ouvertes.fr/hal-03162973/document (cit. on p. 78).

Hong Park, Tae and Dixon Hall (2009). “An Interview with Max Mathews”. In: Computer Music Journal 33.3, pp. 9–22. URL: http://direct.mit.edu/comj/article-pdf/33/3/9/1855364/comj.2009.33.3.9.pdf (cit. on p. 45).

HundredRabbits (2021). 100R — tools ecosystem. URL: https://100r.co/site/tools_ecosystem.html%0A (visited on 10/19/2021) (cit. on p. 30).

Jacobs, Jennifer et al. (n.d.). “Supporting Expressive Procedural Art Creation through Direct Manipulation”. In: (). DOI: 10.1145/3025453.3025927. URL: http://github.com/mitmedialab/para. (cit. on p. 14).

Jensenius, Alexander Refsum and Michael J Lyons (2016). “Trends at NIME-Reflections on Editing ”A NIME Reader””. In: Proceedings of New Interfaces for Musical Expression 2016, pp. 439–443. URL: http://www.nime.org/archive/ (cit. on p. 39).

Jouvelot, Pierre and Yann Orlarey (2011). “Dependent vector types for data structuring in multirate Faust”. In: Computer Languages, Systems & Structures 37.3, pp. 113–131 (cit. on p. 80).

Kay, Alan (1993). “THE EARLY HISTORY OF SMALLTALK”. In: ACM SIGPLAN Notices 28.3, pp. 69–95 (cit. on p. 20).

Kay, Alan and Adele Goldberg (1977). “Personal Dynamic Media”. In: Computer 10.3, pp. 31–41. DOI: 10.1109/C-M.1977.217672. URL: https://dl.acm.org/doi/abs/10.1109/C-M.1977.217672 (cit. on p. 19).

Klabnik, Steve and Nichols, Carol (2020). The Rust Programming Language. URL: https://doc.rust-lang.org/book/ (visited on 06/01/2020) (cit. on p. 65).

Lazzarini, Victor (2013). “The Development of Computer Music Programming Systems”. In: Journal of New Music Research 42.1, pp. 97–110. ISSN: 1744-5027. DOI: 10.1080/09298215.2013.778890. URL: https://www.tandfonline.com/action/journalInformation?journalCode=nnmr20 (cit. on p. 58).

Lieberman, Henry et al. (2016). “End-User Development: An Emerging Paradigm”. In: End User Development. Springer. Chap. 1, pp. 1–8. ISBN: 978-1-4020-4221-8 (cit. on p. 19).

Lopo, Erik de Castro (1990). libsndfile. URL: http://www.mega-nerd.com/libsndfile/ (visited on 05/12/2021) (cit. on p. 67).

Magnusson, Thor (2009). “Of epistemic tools: Musical instruments as cognitive extensions”. In: Organised Sound 14.2, pp. 168–176. ISSN: 13557718. DOI: 10.1017/S1355771809000272 (cit. on pp. 15, 39, 40).

Manovich, Lev (1999). New Media: a User’s Guide. Tech. rep. URL: http://manovich.net/content/04-projects/026-new-media-a-user-s-guide/23_article_1999.pdf (cit. on p. 25).

Mathews, M.V. (1963). “The Digital Computer as a Musical Instrument”. In: Science,New Series 142.3592, pp. 553–557. URL: http://www.jstor.org/stable/1712380 (cit. on p. 45).

Mathews, Max and Curtis Roads (1980). “Interview with Max Mathews”. In: Computer Music Journal 4.4, pp. 15–22 (cit. on p. 45).

Mayer, Mikaël, Viktor Kuncak, and Ravi Chugh (2018). “Bidirectional evaluation with direct manipulation”. In: Proceedings of the ACM on Programming Languages 2.OOPSLA, pp. 1–28. ISSN: 24751421. DOI: 10.1145/3276497. URL: http://dl.acm.org/citation.cfm?doid=3288538.3276497 (cit. on p. 14).

McCartney, James (2002). “Rethinking the computer music language: SuperCollider”. In: Computer Music Journal 26.4, pp. 61–68. ISSN: 01489267. DOI: 10.1162/014892602320991383 (cit. on pp. 56, 76).

McLean, Alex (2014). “Making programming languages to dance to: Live coding with tidal”. In: FARM 2014 - Proceedings of the 2014 ACM SIGPLAN International Workshop on Functional Art, Music, Modelling a New York, New York, USA: Association for Computing Machinery, pp. 63–70. ISBN: 9781450330398. DOI: 10.1145/2633638.2633647. URL: http://dl.acm.org/citation.cfm?doid=2633638.2633647 (cit. on p. 55).

Mclean, Christopher Alex (2011). “Artist-Programmers and Programming Languages for the Arts”. PhD thesis. Goldsmiths, University of London. URL: https://slab.org/writing/thesis.pdf (cit. on p. 15).

McPherson, Andrew and Koray Tahlroğlu (2020). “Idiomatic Patterns and Aesthetic Influence in Computer Music Languages”. In: Organised Sound 25.1, pp. 53–63. ISSN: 14698153. DOI: 10.1017/S1355771819000463 (cit. on p. 57).

Microsoft (2020). Functions - F# — Microsoft Docs. URL: %7Bhttps://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/functions/#function-composition-and-pipelining%7D (visited on 06/06/2020) (cit. on p. 73).

Nishino, Hiroki (2012). “Developing a New Computer Music Programming Language in the ’Research through Design’ Context”. In: SPLASH’12 - Proceedings of the 2012 ACM Conference on Systems, Programming, and 12, pp. 45–47. DOI: 10.1145/2384716.2384736 (cit. on p. 35).

Nishino, Hiroki and Ryohei Nakatsu (2015). “Computer Music Languages and Systems: The Synergy Between Technology and Creativity”. In: Handbook of Digital Games and Entertainment Technologies. ISBN: 978-981-4560-52-8. DOI: 10.1007/978-981-4560-52-8. URL: http://link.springer.com/10.1007/978-981-4560-52-8 (cit. on pp. 42, 52).

Nishino, Hiroki, Naotoshi Osaka, and Ryohei Nakatsu (2014). “LC: A New Computer Music Programming Language with Three Core Features”. In: International Computer Music Conference September, pp. 14–20 (cit. on p. 42).

Norilo, Vesa (2015). “Kronos: A Declarative Metaprogramming Language for Digital Signal Processing”. In: Computer Music Journal 39.4, pp. 30–48. DOI: 10.1162/COMJ_a_00330. URL: https://dl.acm.org/doi/abs/10.1162/COMJ_a_00330 (cit. on p. 80).

– (2016a). “Kronos Meta-Sequencer – From Ugens to Orchestra, Score and Beyond”. In: Proceedings of the International C pp. 117–122 (cit. on p. 69).

– (2016b). “Kronos: Reimagining musical signal processing”. PhD thesis. University of the Arts Helsinki (cit. on p. 76).

Nyquist, H. (1928). “Certain Topics in Telegraph Transmission Theory”. In: Transactions of the American Institute of Elec 47.2, pp. 617–644. ISSN: 0096-3860. DOI: 10.1109/T-AIEE.1928.5055024. URL: http://ieeexplore.ieee.org/document/5055024/ (cit. on p. 44).

Orlarey, Yann, Dominique Fober, and Stephane Letz (2004). “Syntactical and semantical aspects of Faust”. In: Soft Computing 8.9, pp. 623–632. ISSN: 14327643. DOI: 10.1007/s00500-004-0388-1 (cit. on p. 52).

Parks, Lisa and Nicole Starosielski (2015). “Introduction”. In: Signal Traffic: Critical Studies of Media Infrastructures. Board of Trustees of the Univerity of Illinois, pp. 1–27. ISBN: 978-0-252-08087-6 (cit. on p. 81).

Puckette, Miller (2015). “The Sampling Theorem and Its Discontents”. In: International Computer Music Conference, pp. 1–14. URL: http://msp.ucsd.edu/Publications/icmc15.pdf (cit. on pp. 44, 79).

Roads, Curtis (2001). コンピュータ音楽: 歴史・テクノロジー・アート. Trans. by 竜也 青柳 et al. 単行本. 東京電機大学出版局, p. 1054. ISBN: 9784501532109. URL: https://lead.to/amazon/jp/ex-mendeley-jp.html?key=4501532106 (cit. on p. 45).

Ruggieri, Cristina and Thomas P. Murtagh (1988). "Lifetime analysis of dynamically allocated objects". In: Conference Record of the Annual ACM Symposium on Principles of Programming Languages. Vol. Part F130193. Association for Computing Machinery, pp. 285–293. ISBN: 0897912527. DOI: 10.1145/73560.73585 (cit. on p. 78).

Ruiz, Leonardo Laguna (2020). Vult Language. URL: http://modlfo.github.io/vult/ (visited on 09/27/2020) (cit. on p. 72).

Scavone, Gary P. (2002). "RtAudio: A Cross-Platform C++ Class for Realtime Audio Input/Output". In: Proceedings of the 2002 International Computer Music Conference. Goteborg, Sweden (cit. on p. 68).

Sorensen, Andrew and Henry Gardner (2010). "Programming With Time Cyber-physical programming with Impromptu". In: Proceedings of the ACM international conference on Object oriented programming systems lang New York, New York, USA: ACM Press. ISBN: 9781450302036 (cit. on p. 69).

Spinellis, Diomidis (2001). "Notable design patterns for domain-specific languages". In: Journal of Systems and Software 56.1, pp. 91–99. ISSN: 01641212. DOI: 10.1016/S0164-1212(00)00089-3 (cit. on p. 52).

Sumii, Eijiro (2005). "MinCaml: A simple and efficient compiler for a minimal functional language". In: FDPE'05 - Proceedings of the ACM SIGPLAN 2005 Workshop on Functional and Declarative Programming in Educa New York, New York, USA: ACM Press, pp. 27–38. ISBN: 1595930671. DOI: 10.1145/1085114.1085122. URL: http://portal.acm.org/citation.cfm?doid=1085114.1085122 (cit. on p. 68).

Taha, Walid and Tim Sheard (1997). "Multi-Stage Programming with Explicit Annotations". In: SIGPLAN Notices (ACM Special Interest Group on Programming Languages) 32.12, pp. 203–214. ISSN: 03621340. DOI: 10.1145/258994.259019. URL: https://dl-acm-org.anywhere.lib.kyushu-u.ac.jp/doi/abs/10.1145/258994.259019 (cit. on p. 79).

Tanaka, Atau (2010). "Mapping Out Instruments, Affordances, and Mobiles". In: Proceedings of the 2010 Conference on N pp. 88–93 (cit. on p. 39).

カスコーン, キム (2005). "失敗の美学 現代のコンピュータ音楽における「ポスト–デジタル」的傾向". In: ユリイカ 特集＊ポスト・ノイズ 越境するサウンド. Trans. by 順子 長壁. Vol. 37. 58-69 (cit. on p. 25).

ポップ, マーカス (2001). "オヴァルプロセス講義 オヴァル的メソッドとその音楽的アプローチ". In: ポスト・テクノ (ロジー) ミュージック—拡散する「音楽」、解体する「人間. 大村書店. Chap. 2–5, pp. 220–243. ISBN: 4756320260 (cit. on p. 18).

久保田, 晃弘 (2020). Make: Japan — ものをつくらないものづくり 1 — 『Handmade Electronic Music』から再考する「手作 URL: https://makezine.jp/blog/2020/08/make_without_making_01.html (visited on 10/19/2021) (cit. on p. 28).

篠田, ミル (2018). "MIDI 規格の社会史". MA thesis. 東京大学 (cit. on p. 21).

松浦, 知也 (2019). "メディア考古学的視点から音の生成を再考する 3 つの作品制作". MA thesis. 九州大学大学院芸術工学府 (cit. on p. 14).

松浦, 知也 and 一裕 城 (2019). "計算機による音生成の異なるあり方を探る『Electronic Delay Time Automatic Calculator』の制作". In: 先端芸術音楽創作学会 会報. 先端芸術音楽創作学会, pp. 43–49. URL: http://data.jssa.info/paper/2019v11n01/9.Matsuura.pdf (cit. on p. 14).

松浦知也, 城一裕, et al. (2019). "音楽プログラミング言語のソースコードを楽譜と捉え, それを編集するツールの構想". In: 研究報告音楽情報科学 (MUS) 2019.17, pp. 1–4 (cit. on p. 15).

田中, 治久 (hally) (2017). チップチューンのすべて All About Chiptune: ゲーム機から生まれた新しい音楽. 単行本. 誠文堂新光社, p. 335. ISBN: 978-4416616215. URL: https://lead.to/amazon/jp/?op=bt&la=ja&key=441661621X (cit. on p. 42).

馬, 定延 (2014). 日本メディアアート史. アルテスパブリッシング, p. 368. ISBN: 978-4865591163 (cit. on p. 13).