2019 情財第 261 号

2019 年度未踏 IT 人材発掘・育成事業
プログラマブルな音楽制作ソフトウェアの開発
成果報告書

クリエータ　　：松浦 知也
担当 PM　　：竹迫 良範

2020 年 3 月 6 日

目次

1. 要約

本プロジェクトでは、新しい音楽プログラミング言語“mimium”の設計及びその処理系の開発を行った。mimium は音声信号処理のような低レイヤーの処理から、楽譜のような高レイヤーの制御処理を一つの言語体系の中で完結に記述でき、その上で JIT コンパイラによる高速な処理を実現するものである。

2. 背景及び目的

2. 1. 音楽プログラミング言語

音楽に特化したプログラミング言語は 1957 年の MUSIC 1 に始まり、今日に至るまで様々な種類のものが開発されている。図 1 はこれまでに開発されている主要な音楽プログラミング言語を時間軸上に並べたものである。

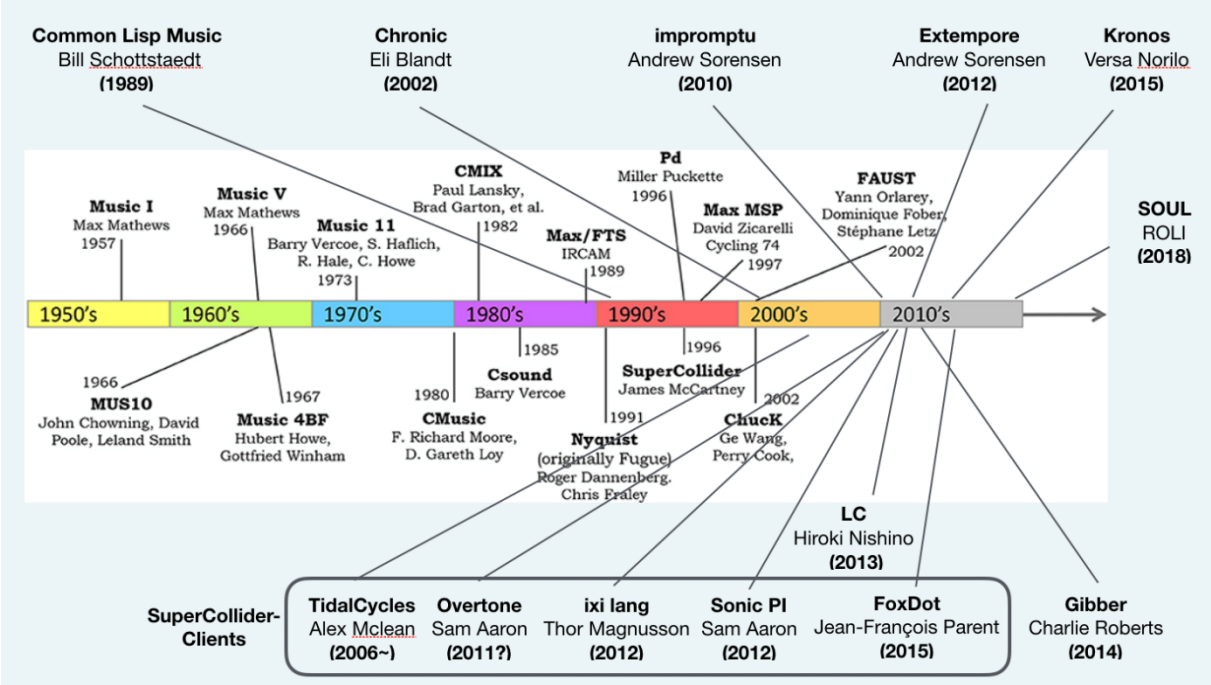

図 1：音楽プログラミング言語の年表。Languages for Computer Music, Roger B. Dannenberg, Frontiers in Digital Humanities, 30 Nov. 2018 から引用、松浦が加筆。

音楽に特化したプログラミング言語を開発する意義として以下のようなものが挙げられる。

(1) 五線譜のような楽譜をコンピュータ上でデジタルデータとして扱えるようにすること。これは Music Macro Language[1] （MML）や MusicXML[2] に代表される音楽マークアップ言語の目的にも近いが、五線譜にも例えば「特定の 8 小節を 2 回繰り返した後最初に戻り、

[1] Music Macro Language：https://ja.wikipedia.org/wiki/Music_Macro_Language
[2] MusicXML：https://www.musicxml.com/

もう一度同じ場所まで来たら別の部分へジャンプする」といった条件によって分岐する表記があることを考えると、五線譜を扱うマークアップ言語も抽象度は低いものの一種のプログラミング言語として扱うことができるだろう。

(2) アルゴリズミックミュージックを記述できるようにすること。例えば特定の確率分布に従ってランダムにフレーズを遷移していくようなケースが考えられる。こうした需要はもともと現代音楽の文脈で利用されることが多かったが、現在ではゲーム内の BGM がユーザーの居場所や行動によってインタラクティブに変化していくものを Wwise[3]に代表されるオーディオミドルウェアを用いて制御するケースなど、ポピュラー音楽の拡張的な形態としても利用されつつある。

(3) 音色のような精密な操作を音楽の一要素として扱うようにできること。2 のケースとも似ているが、例えば確率に応じて曲が遷移するものは人間がサイコロを振って実行することも不可能ではない。しかし数ミリ秒単位で音圧の値を正確に制御することを音楽作品の要素として扱いたい場合にはコンピュータの力に頼らざるを得なくなる。

(4) 一般的な楽譜と異なる音楽の記法を作る事。例えば TydalCycles[4]という言語は複雑なリズムパターンを記述することに特化しており、五線譜では表記が難しいポリリズム（3 拍子と 4 拍子など複数のリズムが混ざったリズム）なども簡潔に記述できる。

(5) コーディングそのものを新しい演奏のインターフェースとすること。ライブコーディングという演奏時にソースコードをその場で書き換え実行していくことでリアルタイムに音楽を作り上げる表現は、近年 Algorave[5]というダンスミュージックをライブコーディングで演奏するイベントの隆盛とともに音楽プログラミング言語の大きな利用の場となっている。

(4)に関してはやや特殊なケースではあるが、(1)〜(5)までをまとめると、音楽プログラミング言語の意義は音、音楽の新しい抽象化と共有の方法をもたらすことにある、と言えるだろう。

2．2．　インフラストラクチャー的な役割を果たす言語

上記の目的とは少し異なる観点で開発されている言語として Faust[6]と SOUL[7]という 2 つの言語がある。

Faust と SOUL は楽譜レベルの処理ではなく音声の信号処理に特化した言語である。どちらも簡潔に信号処理を記述できる一方で C 言語や C++で書く場合と同じか、あるいはそれ以上の実行時性能を発揮することができるという特徴と、同じソースコードがクロスプラットフォームのアプリケーションとして、さらには Web ブラウザ上での実行や組込機器上への埋め込みなど幅広い環境で動作させることができるという特徴がある。

もともと Max[8]や SuperCollider[9]といった古典的な言語では発振器やフィルタといった基本的

[3] Wwise：https://www.audiokinetic.com/ja/products/wwise/
[4] TydalCycles：https://tidalcycles.org/
[5] Algorave：https://algorave.com/
[6] Faust：http://faust.grame.fr/
[7] SOUL：https://soul.dev/
[8] Max：https://cycling74.com/
[9] SuperCollider：https://supercollider.github.io/

な部品（一般的に Unit Generator、UGen と表現される[10]）は C 言語や C++など汎用的な言語で実装されており、ユーザーは音楽に関わる処理だけに注力できる一方で、数多ある用意された UGen の仕様を網羅的に把握しなければいけないという問題がある。また C++を理解していない人にとって UGen の中身はブラックボックスであり、ユーザーが UGen の中身を書き換えることや、派生したものを作るためのハードルが高いといった問題もある[11]。

また、こうした UGen を組み合わせた信号処理を実行する際は各 UGen で処理を行い一時変数に書き込んでから次の処理に渡す、という形を取るためメモリの読み書きコストがかかるほか、複数の処理を合成して最適化を行うことが難しい。

こうした問題に対して、例えば Faust は 1 つのソースコードから Max 用の UGen と SuperCollider 用の UGen を同時に出力できる上、複数の信号処理をコンパイル時に合成して最適化してくれるため、実行時性能もよくなり、その上で処理のブラックボックスが存在しない。

このように 1 つの言語体系でブラックボックスなしで処理を記述でき、複数の環境にまたがって実行可能であるという 2 つの特徴を満たした言語はハードウェアとソフトウェア、あるいは Web ブラウザといった異なる実行環境で使用できる共通基盤としての役割を持つ。また、音楽は音楽家が、それ以下の信号処理のように込み入った低いレイヤーの処理はエンジニアがといった異なる興味の対象を持つ人同士をつなぎ合わせる役割を持っていると言えるだろう。ここでは音楽プログラミング言語におけるこうした特性をインフラストラクチャー的な役割と呼称する。

2．3．　本プロジェクトの目的

本プロジェクトの目的は、Faust や SOUL のようにインフラストラクチャー的な特性を有しつつ、信号処理だけでなく楽譜レベルの制御処理も記述可能な音楽プログラミング言語の設計と、その実行環境（コンパイラ、ランタイム）を開発することである。

楽譜レベルの処理も書けるインフラストラクチャー的な言語が実現されることによって、音楽を一つの固定された配列データではなく、プログラムとして動的に変化しうるものとして制作し、そして配布されることが当たり前になるような未来を実現するための足がかりになる。

プログラムとして音楽を配布する先例としてアンビエント音楽家である Brian Eno がプログラマの Peter Chilvers とともに制作した iPhone アプリとしての音楽作品“Bloom[12]”と“Reflection[13]”が挙げられる。この作品は電子音がプログラムによって動的に生成されることによって無限の再生時間を持ち、再生するたびに異なる音を発する。

音楽をプログラムという形で配布することが当たり前になれば音楽作品自体もこのように動的に変化することが当たり前になり音楽表現の幅がより広がるだろう。

また、音楽をソースコードという形で配布するようになればリミックスや二次創作といった行為もより身近なものになり、音楽作品における「完成」の定義を大きく変えることにもなるだろう。

---

[10] Unit Generator の歴史については [10]に詳しく解説されている。

[11] Brandt はこの問題意識から Chronic という OCaml で時間を扱う型コンストラクタのライブラリを作成しているものの、実行速度に難があるという問題点を挙げている [11]。

[12] Bloom：https://apps.apple.com/jp/app/bloom/id292792586

[13] Reflection：https://apps.apple.com/jp/app/brian-eno-reflection/id1180524479

加えて、近年では音楽のフォーマットはステレオ 2ch や 5.1ch サラウンドにとどまらず、7.1ch、9.1ch、11.1ch、22.2ch サラウンドフォーマットのように非常に多数のスピーカーを用いるもの[14]や、Dolby Atomos[15]のような異なるスピーカー配置に対応するために音源の位置とその移動を記録したオブジェクトベースのフォーマットなどが登場しているが、音楽をソースコードという状態で制作、配布することでこうした特殊なフォーマットでの聴取を特定のプラットフォーマーに依存することなく配布、共有することが可能になるほか、極端に多いスピーカーフォーマットでの利用の場合データサイズの肥大化を防ぐことにもつながる。

3. プロジェクト概要

図 2：mimium のロゴ。
（インフラストラクチャーというコンセプトから横断歩道をイメージして作成した）

本プロジェクトでは、音、音楽を簡潔に記述でき、かつ高い実行時性能を持ち、多様なプラットフォームで動作する音楽プログラミング言語“mimium” （図 2）の設計とその実行環境の開発を行なった。mimium(MInimal Musical medIUM の略として命名)は「音、音楽のためのインフラストラクチャーとしてのプログラミング言語」というコンセプトのもと開発した。mimium は発振器やフィルタのような低レベルの信号処理をゼロから書くことができ、バックエンドにコンパイラ基盤 LLVM（4. 4. で説明）を採用したため C 言語や C++で書く信号処理と遜色ない実行時性能を持つ。また、関数を時間指定して実行することで楽譜レベルの制御処理も行える。この時間指定実行のスケジューリングはオーディオドライバのクロックを元にして行なっているのでサンプル単位での正確さが保証されている。

現在配列や構造体、モジュール機能などを省いた状態で、ベータ版としてとして Mozilla Public License2.0[16]のもと公開されている。現在は macOS をサポートしている。

---

14 放送規格の国際標準化団体 ITU の勧告 BS. 2051-2 を参照。https://www.itu.int/rec/R-REC-BS.2051/en
15 Dolby Atomos：https://www.dolby.com/jp/ja/technologies/dolby-atmos.html
16 Mozilla Public License2.0：https://www.mozilla.org/en-US/MPL/2.0/

4. 開発内容

4. 1. 開発内容

mimium のコンパイラ、ランタイムは主に C++を用いて開発された。
C++を採用した大きな理由は、未踏期間中に導入するかは未定であったものの、当初から LLVM をバックエンドとして採用する予定であり、LLVM の API[17]が用意されている言語（C++，Go，OCaml，C 言語、その他各言語における C 言語 API のラッパー等）の中で提案者が最も使い慣れている言語だったこと、emscripten[18]というツールを用いることで、将来的に Web ブラウザ上に移植する事が容易だと考えたためである。

また C++は 98，03，14，17，20 など幾つかのバージョンが存在するが、本プロジェクトでは比較的モダンかつほとんどの主要なコンパイラで機能の実装が完了している C++17 を採用した。これはコンパイラを実装する上で C++17 から標準ライブラリに導入された std::variant（型安全な共用体）が有用であったことが主な理由である。

また、はじめから最低限 Windows， macOS， Linux をサポートすることを決めていたため、ビルドシステムに CMake[19]を採用した。ビルド過程をできるだけシンプルに保つために、外部ライブラリは最低限のものにとどめたが、コンパイラにはトークナイザとして flex[20]、パーサジェネレータとして bison[21]を利用している。またランタイムの中でクロスプラットフォームのオーディオドライバ RtAudio[22]とオーディオファイル読み込みのための LibSndFile[23]を使用している。

4. 2. プロトタイピング

開発序盤の段階では音楽プログラミング言語特有の機能であるタスクスケジューラのモデルがうまく機能するかどうかを先に確かめることを優先し、パフォーマンス上の問題があった場合は VM（仮想機械）[24]を用いた実装をするという方針をとった。そのためまずソースコードを抽象構文木に変換し、構文木を直接実行する動的型付けのインタプリタ型言語として実装した。

採択前後から 7 月 23 日のブースト会議の時点でこの実装は概ね完成し、サンプル単位で関数の実行が可能であることが確かめられた。またリアルタイム信号処理の機能も簡易的に実装したものの、オーディオ処理で最も基本的な関数であるサイン波を出力するコードだけでも CPU の使用率が 70%を超えてしまうことからこのままでは実用に堪えないことが確認された。この効率の悪さの理由は、毎サンプルの処理毎に構文木へのアクセスを行っていたことや、変数の読み書

---

17 API：Application Programming Interface。ここでは単純にプログラムを用いて制御するための手段という意味合いで使っている。
18 emscripten：https://emscripten.org/
19 Cmake：https://cmake.org/
20 flex：コンパイラの一番入り口で、文字列を予約語などの記号や単語ごとのリストデータに分解するプログラム。https://github.com/westes/flex
21 bison：EBNF という文法規則の記述からパーサープログラム（トークナイザからのデータを受け取り抽象構文木という木構造データを作る処理）を出力するプログラム。コンパイラコンパイラなどとも呼ばれる。https://www.gnu.org/software/bison/
22 RtAudio：https://www.music.mcgill.ca/~gary/rtaudio/
23 LibSndFile：http://www.mega-nerd.com/libsndfile/
24 プログラム上で仮想的な計算機をエミュレーションする方式。

きを単にハッシュテーブル（C++標準ライブラリの std::unordered_map）を使用していたために実行コストがかかっている事などが挙げられた。

ブースト会議ではそもそもリアルタイムのオーディオ処理がどの程度必要であるのかについてのコメントを頂いた。

この時点では未踏期間中は楽譜レベルの処理に限定する、または処理が遅くともオーディオファイル書き出しに特化する事で実現できればそれでも良いのではないかという案もあった。

このうち後者のオーディオファイル書き出し機能についてはブースト会議直後に簡易的に実装したものの、やはり 5 秒の音声ファイル作成に 10 秒以上かかる状態だったので、肝心のオーディオ処理の中身を書いては 10 秒待つという繰り返しに時間をかけるのは非効率的過ぎて中身のコーディングに集中することは難しい状況であった。このためいずれにしても何らかの最適化を行う必要があるという結論に至った。

4．3．　開発方針の確定

9 月の時点で当初想定していた言語とエディタの両方のプロトタイピングから、言語開発に注力することを決定した（7 章参照）。そして開発する言語に求められる機能や他言語と比べた特徴を明確化させた。10 月時点で「インフラストラクチャーとしての音楽プログラミング言語」という最終的に使われる言語のコンセプトが固まった。このコンセプトは下記 3 つの要素を一言で言い表す言葉として用いている。

- 楽譜レベルと信号処理レベルの記述が一つの言語体系で書け、リアルタイム実行できる事
- C や JavaScript といった汎用言語のシンタックスからそれほど逸脱せず、読みやすい言語である事
- 将来的にネイティブアプリケーションとしてだけでなく、ブラウザ上での実行や、マイクロコントローラ上での実行など、多様なプラットフォームで動作する事

この方針のため、未踏期間中にリアルタイムでの信号処理を実現することを目指すため、また後々のプラットフォームの拡大を容易にするために、LLVM というコンパイラ基盤を導入することに決定した。

4．4．　LLVM

LLVM はプログラミング言語の開発を容易に、かつ実行時に最適なパフォーマンスを発揮できるように作られたフレームワークである [1]。LLVM を用いたプログラミング言語はソースコードを一度専用の中間表現（LLVM IR）を介して、各 OS や CPU のためのアセンブリコードを出力するという形をとる。

プログラミング言語開発において LLVM を導入するメリットとして、言語の開発者は LLVM IR の出力部分（フロントエンド）を 1 つ書くだけで幅広い OS や CPU に対応することができる。また、LLVM IR の中で多数のあらかじめ用意された最適化パス（ミドルエンド）を通すことで実行性能を大幅に改善することができる。最適化は言語実装の中でも重要かつ時間のかかる仕事であるため、この部分を LLVM が肩代わりしてくれることで開発者は言語のデザインに注力することが可能になる。

さらに LLVM ではメモリ上で LLVM IR からネイティブコードを出力し、即時実行するエンジンを利用することが可能であるため、ライブコーディングのように動的にコードを評価していく場合にも C 言語で書かれたソースコードと遜色ないパフォーマンスを発揮することができるようになる。

LLVM は C/C++のコンパイラの一つである Clang[25]や、Rust[26]、Julia[27]といったパフォーマンスを意識した汎用プログラミング言語のコンパイラで利用されているほか、Faust や Extempore[28] （の中の xtlang）、Kronos[29]といった 2000 年以後に開発され始めた音楽/音声処理プログラミング言語のバックエンドとして採用されている。

4．5．　MIR（中間表現）の導入

Rust や Julia といったモダンな言語のコンパイラでは、抽象構文木という木構造のデータから LLVM IR を出力するのにはデータ構造に差があり過ぎるため、さらに間に一度中間表現（呼び方はそれぞれ異なるがここでは MIR と呼ぶ）を経由してから出力している [2] [3]。
抽象構文木と MIR の違いは、例えば

```
hoge = 1+ 2 * 3;
```

という式を

```
tmp = 2 * 3;
hoge = 1 + tmp;
```

に変換するような、演算を 1 つ 1 つに分解し一次変数に保存していくことでデータ構造を木構造ではなく命令のリストとして扱うことである。

また、関数が自分の関数の外側で定義された変数（自由変数）を用いることができる（レキシカルスコープを持つ）言語の場合は、自由変数を関数の引数として渡す形に変換する処理、クロージャ変換を行う必要があり、これを中間表現の中で変換するか、さらに一段新しい中間表現を導入することによって解決する。

mimium では 1 段の中間表現を持ち、その中でクロージャ変換を行い LLVM IR を出力する形をとった。この実装には関数型言語の OCaml のサブセットである MinCaml の実装 [4]と、mincaml コンパイラを、LLVM をバックエンドとして Go 言語で実装した gocaml [5] [6]を参考にした。

---

[25] Clang：https://clang.llvm.org/
[26] Rust：https://www.rust-lang.org/
[27] Julia：https://julialang.org/
[28] Extempore：http://digego.github.io/extempore/
[29] Kronos：https://kronoslang.io/

4. 6. mimium の言語仕様

以下に最終的に確定した mimium のソースコードの一例を挙げる。

```
fn countup(active){
    return if (active) (self+1) else 0
}
```

mimium は静的型付けの言語であり、OCaml などの言語で採用されている Hindley-Milner 単相型推論 [7] [8]を備えている（推論に失敗した場合は float に置き換えられる）。引数や返り値の型は以下のように明示的な指定もできる。

```
fn countup(active:float)->float {
    return if (active) (self+1) else 0
}
```

プリミティブ型としては float（64bit 浮動小数点）を唯一の数値型として持ち、他に文字列を扱う string 型が現在実装されている。

4. 7. mimium の言語仕様 - 信号処理

mimium では dsp という名前の関数を作ると、その関数がオーディオドライバによって定期的に呼び出され、その返り値が音圧の値として扱われる[30]。以下に例を示す。

```
//noise.mmm
fn dsp(time:float)->float{
    return random()
}
```

ここで random 関数は組込の-1~1 の範囲でランダムな数を返す組み込みの関数である。
この関数をターミナルより以下のように実行する。

```
mimium noise.mmm
```

このファイルを実行するとホワイトノイズが出力される。

[30] dsp 関数の引数である time はランタイムの現在時刻を表すものだったが、後に専用の now というグローバル予約語が導入されたため、今後削除される予定である。

4. 8. mimium の言語仕様 - self によるフィードバックの表現

mimium の言語としての大きな特徴として、self という予約語を関数内で用いることによって、その関数が最後に出力した値を取得することができる機能がある。この機能により例えば 0~1 までの値を周期的に出力する、ノコギリ波の波形（図 3 参照）を出力するコードを以下のように書くことができる。

```
//phasor.mmm
fn phasor(freq:float){
    return (self+freq/48000)%1
}
fn dsp(time:float)->float{
    return phasor(440)
}
```

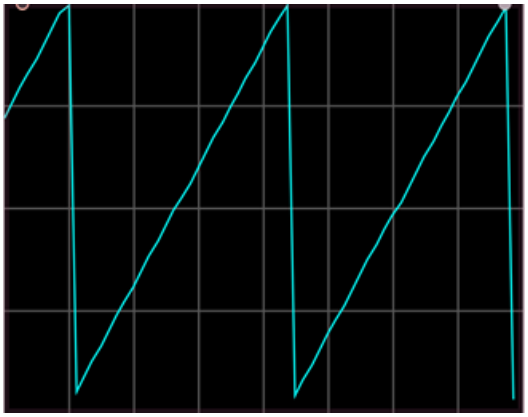

図 3：ノコギリ波の波形。この図では下端が 0、上端が 1 となっている。

self のメモリ確保と読み書きはコンパイラが自動で行う。self を用いた関数、また self を用いた関数を呼び出す関数は内部状態を保持しているので、通常は関数オブジェクトやクロージャのような値と関数の組として表現するが、mimium ではただの関数として書くことが可能になる。

この方法を採用するメリットとして、関数呼び出しをデータフロー型言語のような信号の流れとして表現でき、直感的にわかりやすいという点がある。先程の phasor 関数を使ってサイン波[31]を出力する osc 関数を作る例を考える。

```
fn scaleTwopi(input){
    return input* 2 * 3.141595
}
fn osc(freq){
    return  cos(scaleTwipi(phasor(freq)))
}
```

[31] 三角関数に従う周期関数。あらゆる音を複数のサイン波の合成として表す事ができるため音の基礎単位として扱われ、純音とも呼ばれる。

ここで、c(b(a))というネストした関数呼び出しを a |> b |> c と書くことができる。
パイプライン演算子を導入すると、osc 関数は以下のようにも書ける。

```
fn osc(freq){
    return freq |> phasor |> scaleTwopi |> cos
}
```

同じことをデータフロー型のグラフィカルプログラミング環境である Max を使って書いてみると図 4 のようになる。Max でデータが上から下に流れている表現と同じように mimium でも左から右にデータが流れていく様子が表現できていることがわかる。音楽プログラミング言語におけるデータフローの表現は、現実のシンセサイザーのモジュールやエフェクターを繋いでいく表現と似通っていることもあり、音楽家にとって直感的な表現だと言える。また self でフィードバックを表現するのは Faust からヒントを得た。Faust では、内部状態を持つ関数を表現するのにフィードバック演算子(~)という専用の演算子を用いている。phasor 関数を Faust で書くと以下のようになる。

```
phasor(freq) = +(freq/48000) ~ %(1);
```

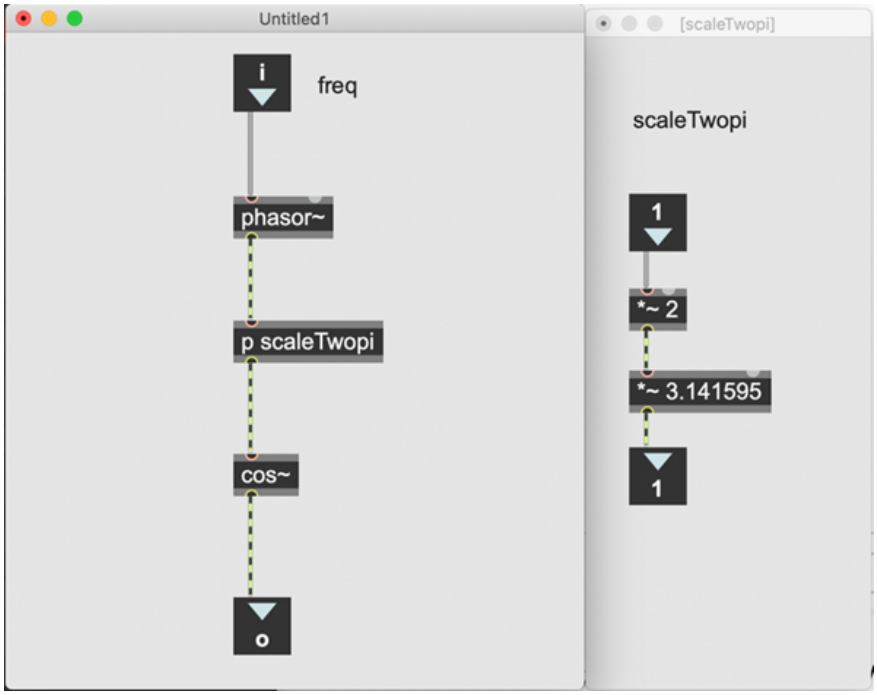

図 4：Max を用いてサイン波を出力するプログラムの例 [32]

この内容であればまだ読むことができるが、Faust で複雑なフィードバックを書くと可読性が落ちてくる。mimium では関数の単位でフィードバックを行うようにすることで、自然と複雑なフィードバック接続を分離して書くことができるような文法になっている。

[32] あくまで比較のために模式的に作成したものであって、[cos~]というオブジェクトははじめから入力範囲が 0~1 になっている。また、そもそもサイン波を出力するためのオブジェクトとして[cycle~]が用意されている。

4. 9. ファイル読み出しの例

mimium では現在組み込み関数である loadwav 関数と loadwavsize 関数を用いてオーディオファイルの読み込みを行うことができる。この 2 つの関数は LibSndFile のライブラリを用いて実装されている。以下にオーディオファイルを読み込みループ再生するコードの例を示す。

```
name = "/Users/tomoya/Music/Chickens.wav"
mysize = loadwavsize(name)
mywave = loadwav(name)
fn seek(speed:float,size:float){
    return (speed+self)%size
}
fn dsp(time:float)->float{
    return mywave[seek(1,mysize)]
}
```

loadwav 関数の返り値は float の配列[33]で、[position]を用いてアクセスすることができる。通常の言語であれば position は整数である必要があるが、mimium には整数型が存在しないため position に浮動小数点を使うことができる。この場合、例えば mywave[1.3]というような呼び出しをすると、線形補完された mywave[1]*0.7 + mywave[2]*0.3 が返り値となる。このように補完がデフォルトで実装されていることで、例えば seek(0.5,mysize)のように再生速度を半分以下にしてもノイズが乗らなくなる。

4. 10. mimium の言語仕様 - 制御処理

楽譜レベルの遅い処理は関数の時間指定実行機能を使うことで実現できる。以下に例を示す。

```
ntrigger = 1
fn setN(val:float){
    ntrigger = val
}
fn playN(duration:float)->void{
    setN(1)@now
    setN(0)@(now+duration)
}
fn Nloop(period:float)->void{
    playN(50)
    nextperiod = if(random()>-0.3) period/2 else period
    Nloop(period)@(now+nextperiod)
}
Nloop(12000)@0
```

[33] 一つのデータ型（ここでは float）の集合がメモリ上に直列に配置されたデータ構造。現時点では mimium で配列を作る事ができるのは loadwav 関数のみである。

関数呼び出しに@マークをつけるとランタイムのタスクスケジューラに関数名と引数、実行すべき時間が登録され、オーディオドライバのクロックを元に指定された時間に関数を実行する。

例えば playN という関数では実行されたタイミングで ntrigger という変数を 1 に書き換え、引数 duration が経過した後に ntrigger を 0 に書き換える。さらに Nloop という関数では playN を実行した後に Nloop 自身を一定時間後に再帰的に呼び出すようになっている。時間指定も変数を使って動的に指定することができるので、上のコードでは乱数を用いて一定の確率で呼び出しの時間を半分にしている。このように時間を跨いだ再帰呼び出しを使う方法は Extempore という Lisp ベースの音楽プログラミング環境で提案された Temporal Recursion というデザインパターン [9]に影響を受けたものである。以下は Extempore で Temporal Recursion を用いたコードの例である。

```
(define foo
  (lambda ()
    (play-note (now) synth 60 80 *second*)
    (callback (+ (now) *second*) 'foo)))
(foo)
```

Extempore では callback という専用の関数を用いることで実現されているが、mimium では@マークを用いて読みやすくなるようにしている。

また、時間を指定した関数呼び出しは評価されたタイミングでは実行されないため、返り値を利用することができない。そのため、信号処理では関数のパイプで返り値を積極的に利用した書き方を行う一方、制御処理は基本的に変数の破壊的代入など副作用を伴う手続型の処理を書くことになる。

4. 11. モジュラーアーキテクチャ

図 5 は mimium の実行環境を図式化したものである。

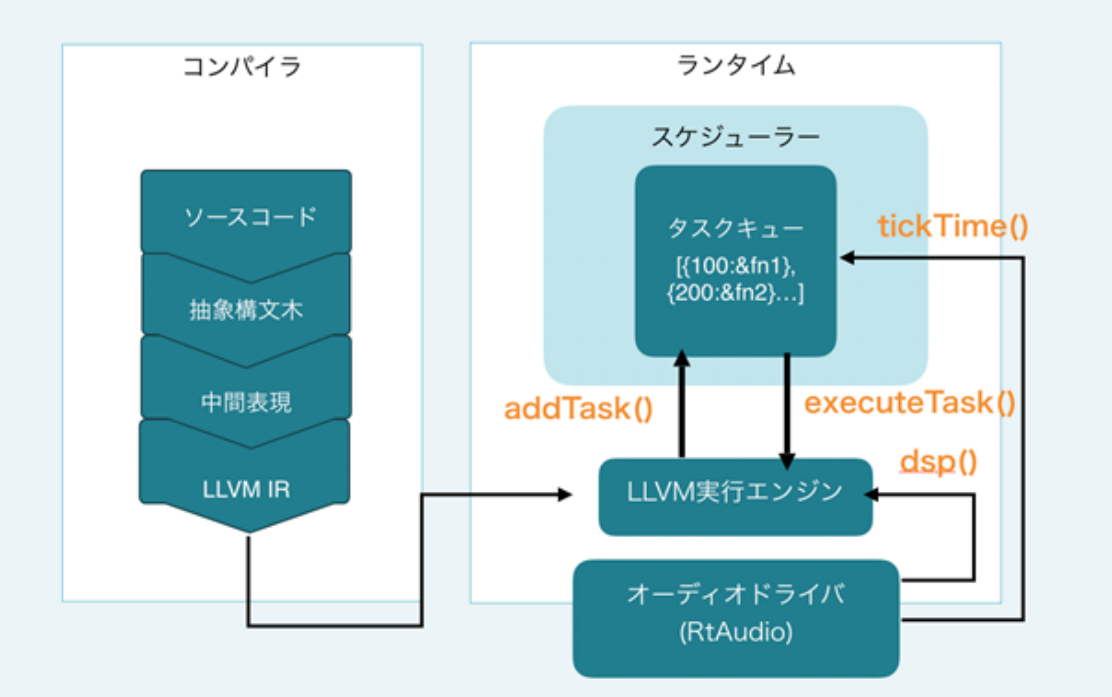

図 5：mimium 実行環境の構成を表したもの

実行環境は主にコンパイラとランタイムの 2 つで構成される。コンパイラはソースコードを読み込み、型推論などを行い、LLVM IR を出力するところまでを担う。

ランタイムは LLVM IR を読み込んでネイティブコードにコンパイルし、実際に実行する。同時にオーディオドライバからのトリガを元に dsp 関数を実行する他、@マークで指定された関数を内部のキューに溜め込み、指定時間になったら実行するタスクスケジューラを内部に持つ。キューは実行時間によってソートされる優先順位つきキュー[34]（C++標準ライブラリの std::priority_queue）として実装されている。

このランタイムの実装は関数の時間指定実行と同様 Extempore のランタイムを参考にして実装した。

コンパイラとランタイムは実装としてそれぞれ独立しているので、例えば LLVM IR を一度ファイルに書き込んで、ランタイムが LLVM IR を読み込み実行するように、分離しての運用も可能である。このように各部の独立性を確保しておくことによって、今後の展開としてオーディオドライバをファイル読み書きのものに差し替えて、大量のオーディオファイルをバッチ処理するツールとしての利用や、オーディオプラグインや Web ブラウザ上への埋め込みなど幅広いプラットフォームでの利用が可能になる。

[34] 先入れ後出し（FILO）データ構造のこと。入力されたデータを前から順番に積んでいき、後ろから取り出すが、優先順位付きキューではデータを積む時点でデータやデータに紐付けたラベル（例えば今回は関数に紐付けた時間）に従って一度ソートしてから取り出す。

4. 12. C 言語ライブラリ連携

mimium では C 言語のライブラリとの連携が比較的簡単に行えるようになっている。三角関数や指数、対数関数といった基本的な数学関数は ffi.cpp というソースの中で（図 6)、その関数の名前、mimium 上での型の情報、実際に呼び出す C 関数の名前を指定することで呼び出しができるようになる。関数の型の指定もコンパイラ内部で用いられている型のコンストラクタをそのまま用いることで簡潔に指定することができるようになっている。

この方式を取ることによって例えば数学関数であれば libc（OS に標準で付属する C 言語ライブラリ）の数学関数を直接指定しているので実行時のオーバーヘッドをなくすことができる。

また、当然自分で定義した C ライブラリの関数も利用することが可能であり、コンソール出力用の println や、オーディオファイル読み込みの loadwav、値を返す if 式、配列アクセスの線形補完などは C++上で実装したものを同様にマップし呼び出せるようにすることで実現している。今後は mimium のソースコード上で同様のことを行うことで、動的にライブラリをロードすることを可能にする機能の実装を予定している。

```
std::unordered_map<std::string, BuiltinFnInfo> LLVMBuiltin::ftable = {
    {"print", FI{Function(Void(), {Float()}), "printdouble"}},
    {"println", FI{Function(Void(), {Float()}), "printlndouble"}},
    {"printlnstr", FI{Function(Void(), {String()}), "printlnstr"}},



    {"sin", FI{Function(Float(), {Float()}), "sin"}},
    {"cos", FI{Function(Float(), {Float()}), "cos"}},
    {"tan", FI{Function(Float(), {Float()}), "tan"}},

    {"asin", FI{Function(Float(), {Float()}), "asin"}},
    {"acos", FI{Function(Float(), {Float()}), "acos"}},
    {"atan", FI{Function(Float(), {Float()}), "atan"}},
    {"atan2", FI{Function(Float(), {Float(), Float()}), "atan2"}},

    {"sinh", FI{Function(Float(), {Float()}), "sinh"}},
    {"cosh", FI{Function(Float(), {Float()}), "cosh"}},
```

図 6：コンパイラの中の ffi.cpp のスクリーンショット

4. 13. シンタックスハイライト

コンパイラとは別に、開発にメインで使用していたエディタである Visual Studio Code 向けのシンタックスハイライトを作成した（図 7)。

最終的な言語仕様が Rust に近いものとなったため、Rust 用の設定ファイルを参考に@, now, self などの mimium 特有の機能を追加している。これは Visual Studio Code のマーケットプレイスからダウンロード可能になっている。

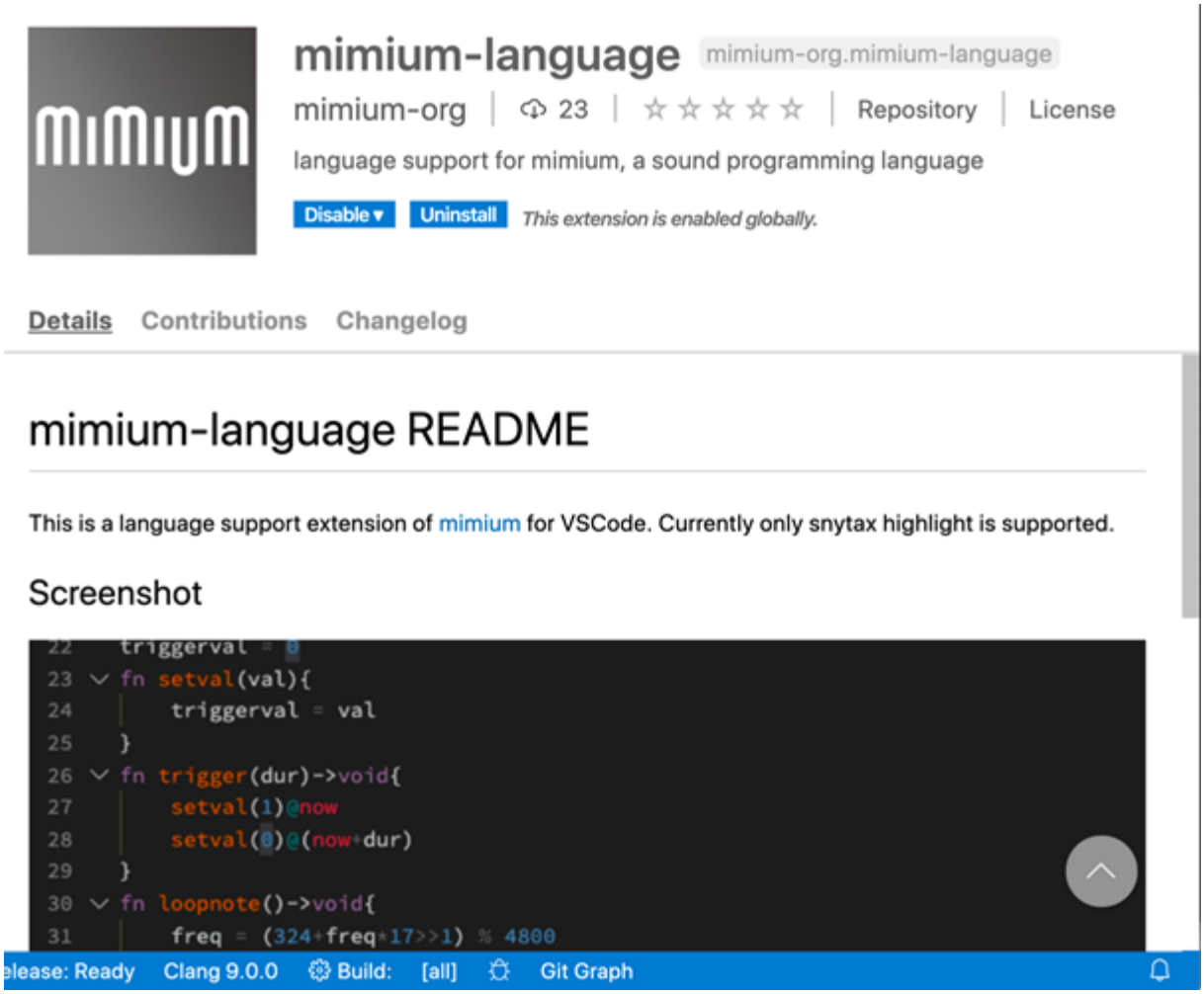

図 7：Visual Code Studio 用の mimium シンタックスハイライト拡張機能スクリーンショット

5. 開発成果の特徴

表 1 は mimium と他言語の特徴を比較したものである。mimium が他の言語と比べて特徴的なのは、下記 3 点を全て満たす部分である。

- 信号処理が記述可能
- 信号処理の記述を基本的な単位からフルスクラッチで、尚且つパフォーマンスを損なわず記述可能
- 制御処理も同じ言語の中で記述可能

先行する Extempore も 3 つ全てを同様に満たしているが、制御処理に関しては Scheme[35]、信号処理に関しては Scheme を拡張した静的型付けの独自に開発された Lisp 系言語の xtlang の 2 つを使い分ける形となっており、mimium はこれを一つの言語体系で完結させているという違いがある。

[35] Lisp 系の言語の中で最も実用的に使われている言語の一つ。動的型付けでありレキシカルスコープを持つ（関数の中で、その外側で定義された変数を使用できる）。http://www.scheme-reports.org/

ChucK では同じ言語を用いて信号処理をフルスクラッチで書く事が可能な拡張機能 ChuGen が導入されているが、これは mimium における@を用いた関数の時間指定実行を 1 サンプルごとに行うような構造をとっているため、実行時性能に難がある。

表 1：mimium と他の言語の特徴比較一覧

	Max	SuperCollider	Extempore	Faust	ChucK	mimium
信号(速い)処理	○(Max)	○(hoge.ar())	○(xtlang)	○	○	○
フルスクラッチ信号処理	△(Gen)	×	○	○	△(拡張)	○
制御(遅い)処理	○(MSP)	○(hoge.kr())	○(Scheme)	×	○	○

6. 今後の課題，展望

mimium の今後の開発予定としては、言語仕様として最低限必要となるであろう、配列と構造体/タプル、加えてモジュール機能の開発と、現在対応している macOS 以外に Windows、Linux、Web ブラウザでの実行に対応を行なった上で正式版のリリースをする予定である。
並行して、利用者及び開発者のコミュニティを広げていくために、macOS 以外の OS への対応と、ドキュメンテーションの整備を進めていく。

長期的な開発目標としては、音楽の制作の部分で積極的に利用してもらうための IDE（Integrated Development Environment）の開発、更には GUI を用いたソースコード編集ツールの開発（後述）を行う他、音楽の配布の形として、Node.js における npm のようなパッケージマネージャの機能拡張として、mimium を使って制作された音楽作品を簡単に共有できるようなプラットフォームの開発を検討している。

また本プロジェクトは学術的にも価値があるものと考えているので、学会での論文投稿も予定している。

7. 実施計画書内容との相違点

本プロジェクトでは当初、一般的に使われている音楽制作ソフトウェアである Digital Audio Workstation ソフトウェア（以下、DAW という）と音楽プログラミング環境の中間的位置づけとして、楽譜のように機能する音楽プログラミング言語を設計し、そのソースコードを DAW のようにグラフィカルに編集することができる、音楽制作ソフトウェアの開発を予定していた。

具体的な開発目標は以下のようになっていた。

(1) 音楽プログラミング言語仕様の策定・技術調査
音楽プログラミング言語の仕様の策定と、そのための技術調査を行う。本言語では、時間の型を持ち、MIDI のようなイベント処理とオーディオのようなストリームの処理を他言

語に頼らず一つの言語体系で記述でき、また冗長性の排除よりもテキストとしての可読性を高め、ソースコードが楽譜のようにも機能することを目指す。

(2) 音楽プログラミング言語の実行環境の開発

(1).で策定した音楽プログラミング言語の実行環境を実装する。本実行環境は、オーディオ処理のためにパフォーマンスが求められることと、LLVM を使用することで異なるハードウェアやプラットフォームでの実行が可能になることから、C++を使った実装を想定する。

(3) ソースコード編集ツールの開発

(1).のプログラミング言語のソースコードを読み込み、その構造を一般的な DAW のように時間軸上に複数のトラックが存在し、その中に時間範囲を指定して配置されたオブジェクトとして表示し、ソースコードと双方向に編集できるソフトウェアを実装する。本編集ツールでは、マウスを用いた波形やリズムパターンのアルゴリズミックな生成を支援する GUI、出力を音声ファイルや MIDI パターンとして書き出す機能などの実現を目指す。

(4) 全体の環境統合・リリース

(1), (2), (3)を統合して本ソフトウェアを β 版として公開する。ユーザフィードバックを集め、それを元に改善を行う。

最終的に本プロジェクトでは、(1), (2)の音楽プログラミング言語が、音楽制作環境のみならず、音楽配信の媒体としても機能するような、将来の音楽制作、聴取環境の更新を試みるものとして重要な位置付けにあるものと捉え、その言語仕様やコンパイラ、実行環境を含めた全体のエコシステムの設計と提案を試みるものとなった。

(3)の GUI を用いてソースコードを編集する音楽制作ソフトウェアの開発は、9 月中間では技術調査を中心に行ない、一部プロトタイピングも行った。

プロトタイプは Web アプリケーションとして実装し、ローカル環境のコンパイラを起動し、ソースコードの抽象構文木を JSON として出力させ、それを JavaScript で編集するような構造を検討していた。このうちコンパイラの JSON 出力機能と、ブラウザを用いたテキストエディタ部分の簡易的な実装などを行った。

しかしながらエディタの実装が言語実装と扱う技術トピックにあまりに違いがあるということと、コンパイラの機能が不十分な状態で GUI の開発を進めると後々整合性が取れなくなる可能性を考えて、言語仕様が確定し、コンパイラの機能が十分に発達した後での開発を行うべきだと判断した。

今後コンパイラの開発を進める中で、最適化やエラー、警告表示の処理に必要なソースコードの静的解析機能が強化され、その延長線上にエディタ上でエラーや文法修正の提案をユーザーにフィードバックする Language Server（Microsoft Visual Studio における IntelliSense にあたるもの）を開発する予定で、さらにその延長線上に GUI を用いて数値を変更し、ソースコードに反映する双方向プログラミングの機能を実装することができ、自然にエディタの開発へとつなげていくことが可能だと考えている。

8. 成長の自己分析

提案者はこれまで言語処理系のような、比較的低いレイヤーでの本格的なソフトウェア開発の経験がほとんどなかったため、継続的な開発環境の整備や、テスト記述、CI の導入など、一般的なソフトウェア開発におけるノウハウを学ぶのに時間を多く取られた部分があった。またコンピ

ューターサイエンス全般の知識をプロジェクト期間中に独学で身に付けながらの開発でもあったので、幾分開発効率は良いとは言えなかったかもしれない。その一方で他プロジェクト（特にセキュリティ関連など低レイヤーなもの）との交流から周辺知識を効率的に得ることができ、完全に独学でプロジェクトを進めていた場合と比べて相当に速く開発を進めることができたと考えている。

またツールをただ作るだけでなく実際現実的にどう使ってもらえるか、社会に浸透させていけるかを他プロジェクトへのものも含めたアドバイスを聞く中で真剣に考えるようになり、ライセンスの選択やオープンソースソフトウェアの持続的な開発の方法についての知見を深めることにつながった。

9. 付録

(1) 関連 Web サイト

筆者個人 Web サイト URL：https://matsuuratomoya.com

開発リポジトリ URL ： https://github.com/mimium-org/mimium

(2) 参考文献

[1] C. Lattner, V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” 30 September 2003. [オンライン]. Available: https://llvm.org/pubs/2003-09-30-LifelongOptimizationTR.html. [アクセス日: 28 February 2020].

[2] The Julia Project, “Julia SSA-form IR,” 4 August 2019. [オンライン]. Available: https://docs.julialang.org/en/v1/devdocs/ssair/. [アクセス日: 28 February 2020].

[3] N. Matsakis, “Introducing MIR - Rust Blog,” 19 April 2016. [オンライン]. Available: https://blog.rust-lang.org/2016/04/19/MIR.html. [アクセス日: 28 02 2020].

[4] 住. 英二郎, “速攻 MinCaml コンパイラ概説,” 27 August 2012. [オンライン]. Available: http://esumii.github.io/min-caml/. [アクセス日: Febrary 2022].

[5] Linda_pp, “Go でつくる汎用言語処理系 実装戦略,” 24 March 2017. [オンライン]. Available: https://speakerdeck.com/rhysd/go-detukurufan-yong-yan-yu-chu-li-xi-shi-zhuang-zhan-lue. [アクセス日: 28 February 2020].

[6] Linda_pp, “gocaml,” 1 May 2018. [オンライン]. Available: https://github.com/rhysd/gocaml. [アクセス日: 28 February 2022].

[7] J. R. Hindley, “The Principal Type-Scheme of an Object in Combinatory Logic,” Transactions of the American Mathematical Society, 第 巻 146, pp. 29-60, 1969.

[8] R. Milner, “A Theory of Type Polymorphism in Programming,” Journal of Computer and System Sciences, 第 巻 17, 第 3, pp. 348-374, 1978.

[9] A. Sorensen, “The Many Faces of a Temporal Recursion,” 2013. [オンライン]. Available: http://extempore.moso.com.au/temporal_recursion.html. [アクセス日: 28 February 2020].

[10] N. Hiroki, N. Ryohey, “Computer Music Languages and Systems: The Synergy Between Technology and Creativity,” 著: Handbook of Digital Games and Entertainment Technologies, Springer, Singapore, 2016.
[11] E. Brandt, “Temporal type constructors for computer music programming,” PhD Thesis, School of Computer Science, Carnegie Mellon University, 2002.