

インフラストラクチャとしての 音楽のためのプログラミング言語 mimumの設計と開発

2021/11/29 松浦知也 博士論文 予備審査 me@matsuuratomoaya.com

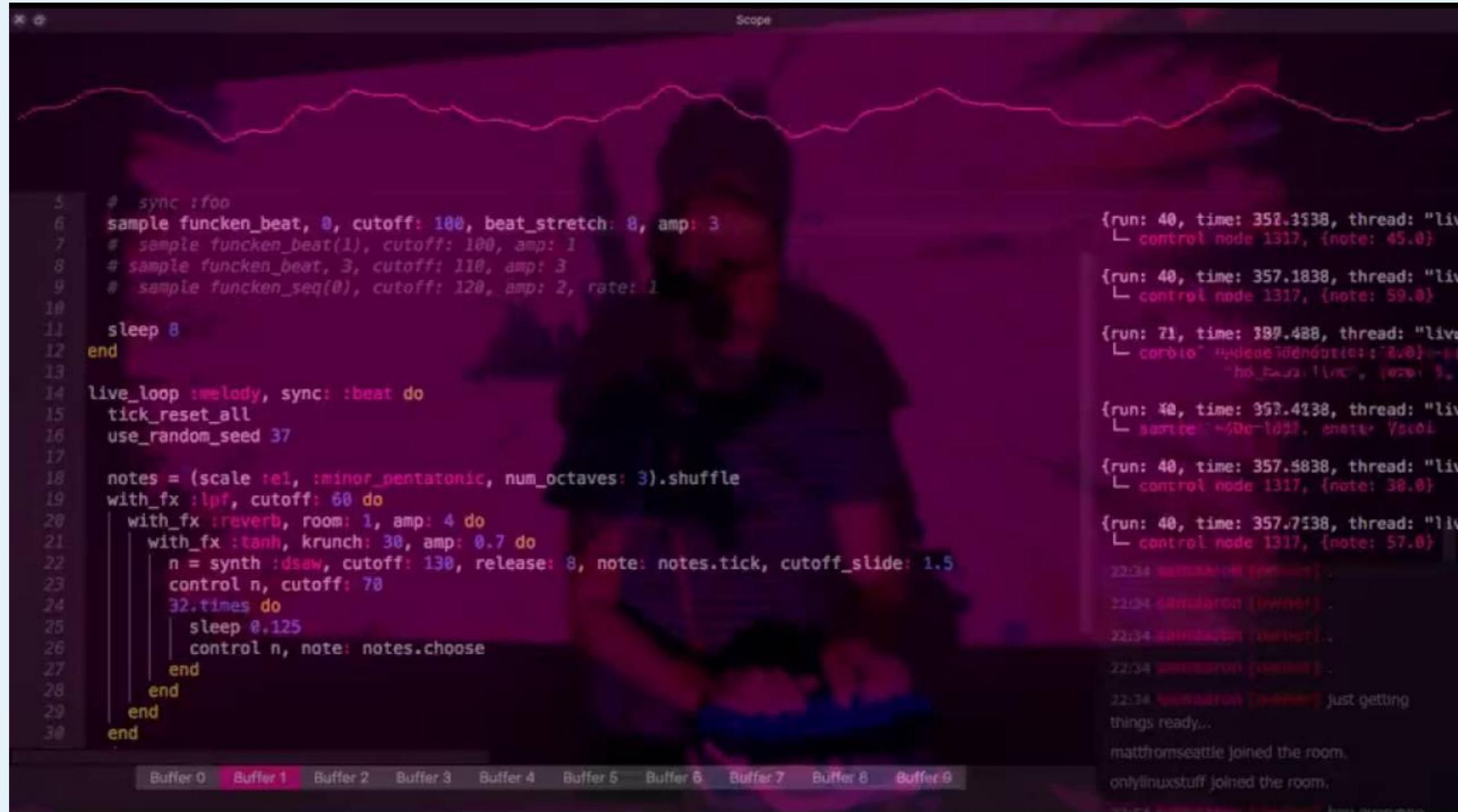
概要

- 音楽のためのプログラミング言語(Programming Language for Music:PLfM)、mimiumの開発を通じて、PLfMとは何かという存在論とその制作行為の歴史的位置付けを整理する。
- その上でmimiumを単なる制作のための道具ではなく、インフラストラクチャとしてのPLfMとして提案する
- 音楽を、テクノロジーの応用分野として捉えるのではなく、音楽のために（とくにコンピューティング）技術の根幹を考え直す、音楽土木工学という学問領域を提示

Chapters

- | | |
|----------------------------|------------------------------|
| 1. イントロダクション | (動機とメタ方法論) |
| 2. デザインリサーチとしてのPLfM研究 | (研究パラダイムの設定) |
| 3. 音楽家の実践としてのPLfM研究 | (研究課題の設定) |
| 4. What1:歴史的に見た音楽プログラミング言語 | (技術要素の通時的整理) |
| 5. What2:音楽プログラミング言語の特性の整理 | (技術要素の共時的整理) |
| 6. 音楽プログラミング言語mimiumの設計と実装 | (暫定的解の提案) |
| 7. 議論 | (暫定解の妥当さを、5~2の視点を逆順にそれぞれに検討) |
| 8. 結論 | (課題と展望としての“音楽土木工学”概念の提示) |

Programming Language for Music



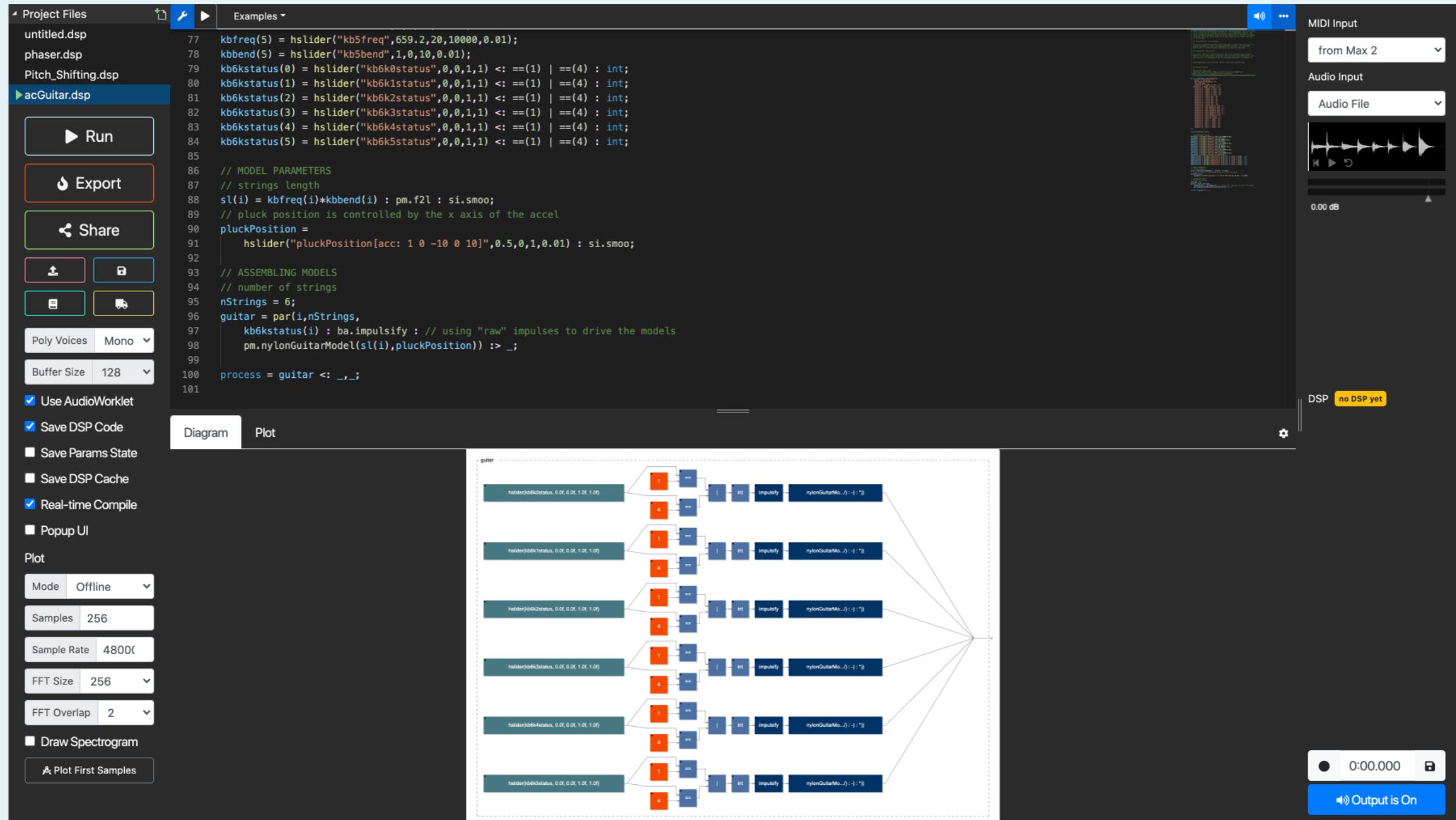
```
5  # sync :foo
6  sample funcken_beat, 0, cutoff: 100, beat_stretch: 8, amp: 3
7  # sample funcken_beat(1), cutoff: 100, amp: 1
8  # sample funcken_beat, 3, cutoff: 110, amp: 3
9  # sample funcken_seq(0), cutoff: 120, amp: 2, rate: 1
10
11 sleep 8
12 end
13
14 live_loop :melody, sync: :beat do
15   tick_reset_all
16   use_random_seed 37
17
18   notes = (scale :e1, :minor_pentatonic, num_octaves: 3).shuffle
19   with_fx :lpf, cutoff: 60 do
20     with_fx :reverb, room: 1, amp: 4 do
21       with_fx :tanh, krunch: 30, amp: 0.7 do
22         n = synth :dsaw, cutoff: 130, release: 8, note: notes.tick, cutoff_slide: 1.5
23         control n, cutoff: 70
24         32.times do
25           sleep 0.125
26           control n, note: notes.choose
27         end
28       end
29     end
30   end
end
```

Buffer 0 Buffer 1 Buffer 2 Buffer 3 Buffer 4 Buffer 5 Buffer 6 Buffer 7 Buffer 8 Buffer 9

{run: 40, time: 352.3538, thread: "live", control node 1317, (note: 45.0)}
{run: 40, time: 357.1838, thread: "live", control node 1317, (note: 59.0)}
{run: 71, time: 387.488, thread: "live", corbie" /ydeue10endowies: "820)-n81 "hd_bura.11mc", (note: 5.0)}
{run: 40, time: 392.4238, thread: "live", sample: >/0e-1d3d, (note: 75.0)}
{run: 40, time: 357.5838, thread: "live", control node 1317, (note: 38.0)}
{run: 40, time: 357.7538, thread: "live", control node 1317, (note: 57.0)}
22:34 ~~matthomseattle~~ (owner).
22:34 ~~onlylinuxstuff~~ (owner).
22:34 ~~matthomseattle~~ (owner).
22:34 ~~onlylinuxstuff~~ (owner).
22:34 ~~matthomseattle~~ (owner) just getting things ready...
matthomseattle joined the room.
onlylinuxstuff joined the room.

Sam AaronによるSonic Piを用いたパフォーマンス (2016)

Programming Language for Music



Project Files

- untitled.dsp
- phaser.dsp
- Pitch_Shifting.dsp
- acGuitar.dsp

Run

Export

Share

Poly Voices Mono

Buffer Size 128

Use AudioWorklet

Save DSP Code

Save Params State

Save DSP Cache

Real-time Compile

Popup UI

Plot

Mode Offline

Samples 256

Sample Rate 48000

FFT Size 256

FFT Overlap 2

Draw Spectrogram

Plot First Samples

```
77  kbfreq(5) = hslider("kb5freq",659.2,20,10000,0.01);
78  kbbend(5) = hslider("kb5bend",1,0,10,0.01);
79  kb6kstatus(0) = hslider("kb6k0status",0,0,1,1) <: ==(1) | ==(4) : int;
80  kb6kstatus(1) = hslider("kb6k1status",0,0,1,1) <: ==(1) | ==(4) : int;
81  kb6kstatus(2) = hslider("kb6k2status",0,0,1,1) <: ==(1) | ==(4) : int;
82  kb6kstatus(3) = hslider("kb6k3status",0,0,1,1) <: ==(1) | ==(4) : int;
83  kb6kstatus(4) = hslider("kb6k4status",0,0,1,1) <: ==(1) | ==(4) : int;
84  kb6kstatus(5) = hslider("kb6k5status",0,0,1,1) <: ==(1) | ==(4) : int;
85
86 // MODEL PARAMETERS
87 // strings length
88 sl(i) = kbfreq(i)*kbbend(i) : pm.f2l : si.smoo;
89 // pluck position is controlled by the x axis of the accel
90 pluckPosition =
91     hslider("pluckPosition[acc: 1 0 -10 0 10]",0.5,0,1,0.01) : si.smoo;
92
93 // ASSEMBLING MODELS
94 // number of strings
95 nStrings = 6;
96 guitar = par(i,nStrings,
97     kb6kstatus(i) : ba.impulsify : // using "raw" impulses to drive the models
98     pm.nyonguitarModel(sl(i),pluckPosition)) :> _;
99
100 process = guitar <: _,_;
```

MIDI Input

from Max 2

Audio Input

Audio File

0.00 dB

DSP no DSP yet

Diagram Plot

0:00.000

Output is On

Faust言語を用いたソフトウェアシンセサイザーの構造的記述

音楽のためのプログラミング言語

Programming Language for Music:PLfM

- 既存の用語...Computer Music Language(CML)/System/Environments
- 必ずしもComputer Musicのためではないが、音楽や音声処理に特化したプログラミング言語、という枠組みが欲しい
- プログラミング言語≠その言語を実行するソフトウェア
- 音楽言語においてはほとんど1言語=1実行ソフトウェアだが、言語仕様が形式的に定義されていれば手でも計算できる(Faustなど)

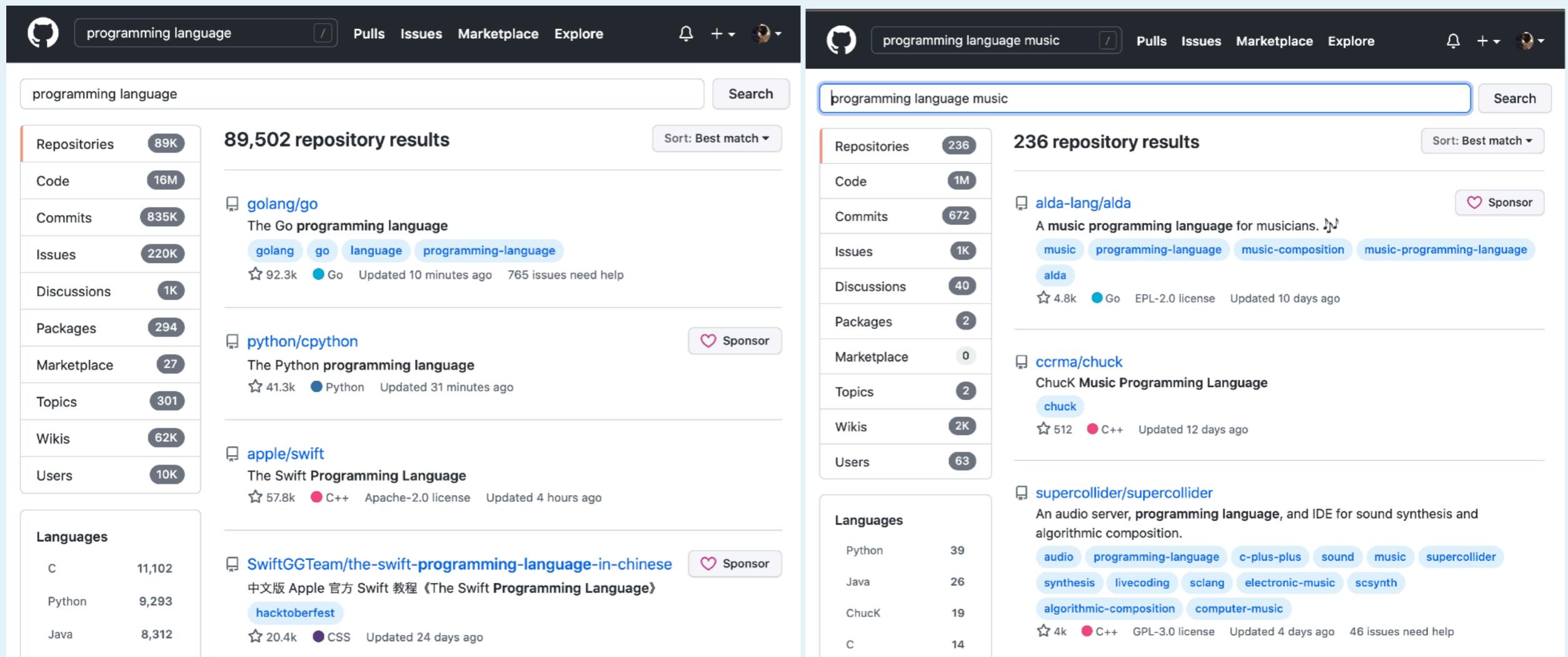
イントロダクション

mimum開発そのものへの動機

- 現在、音楽制作（Digital Audio Workstationソフトウェア）から音楽の聴取（信号処理チップなど）まで、あらゆるところでコンピューターが使われている
- その割には、音楽のフォーマットや聴取の環境そのものは2chステレオから大きくは変化していない
- コンピューターは万能の装置であるはずなのに、なぜ音楽のためのソフトウェアやコンピューターはユーザーがカスタマイズすることが難しいのか
- 音楽のためのプログラミング環境は長い歴史がある割にプログラミングという行為は音楽制作の主要な手段にはなっていないのはなぜか？

イントロダクション

動機



The image shows two side-by-side GitHub search results pages. The left page is for the search term 'programming language', which yields 89,502 results. The right page is for the search term 'programming language music', which yields 236 results. Both pages include a sidebar with repository statistics and a 'Languages' section.

Left (programming language results):

- Repositories:** 89K
- Code:** 16M
- Commits:** 835K
- Issues:** 220K
- Discussions:** 1K
- Packages:** 294
- Marketplace:** 27
- Topics:** 301
- Wikis:** 62K
- Users:** 10K

Right (programming language music results):

- Repositories:** 236
- Code:** 1M
- Commits:** 672
- Issues:** 1K
- Discussions:** 40
- Packages:** 2
- Marketplace:** 0
- Topics:** 2
- Wikis:** 2K
- Users:** 63

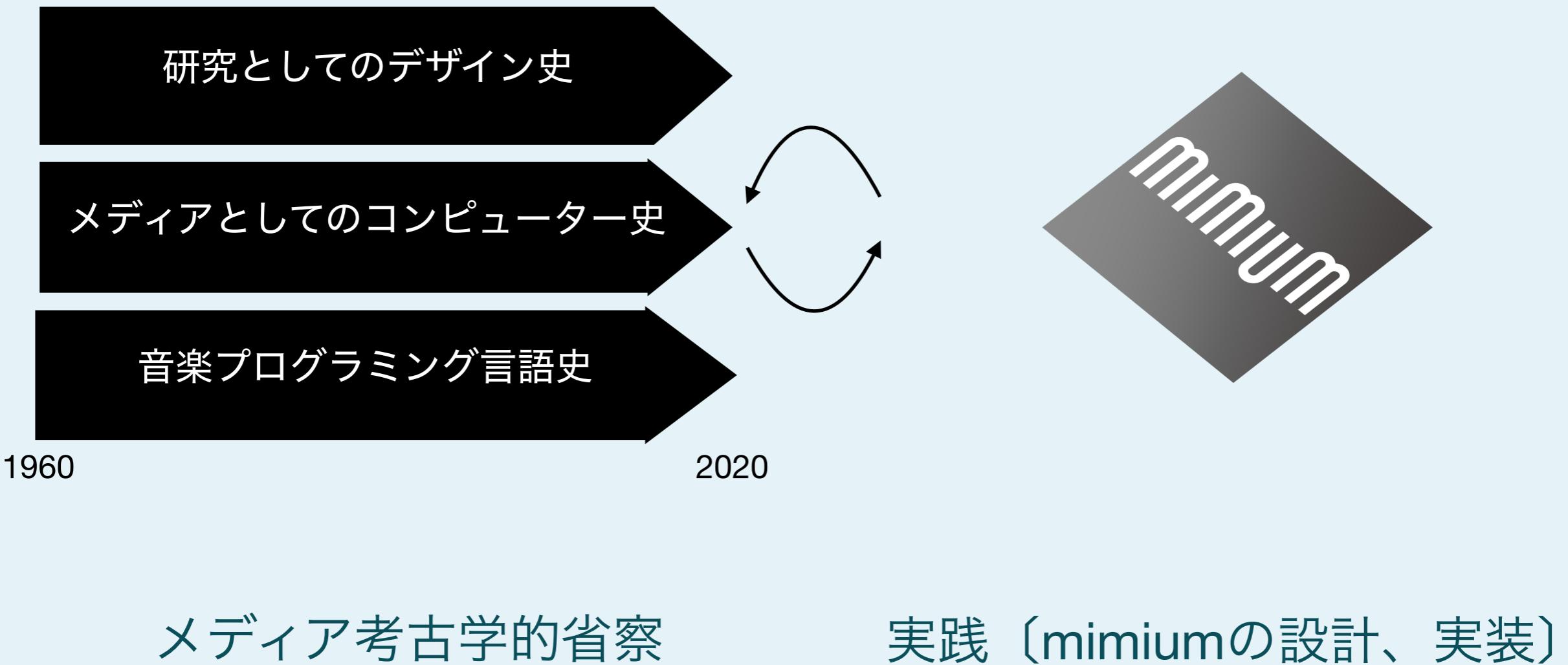
Common Languages (across both pages):

- C: 11,102
- Python: 9,293
- Java: 8,312
- Python: 39
- Java: 26
- Chuck: 19
- C: 14

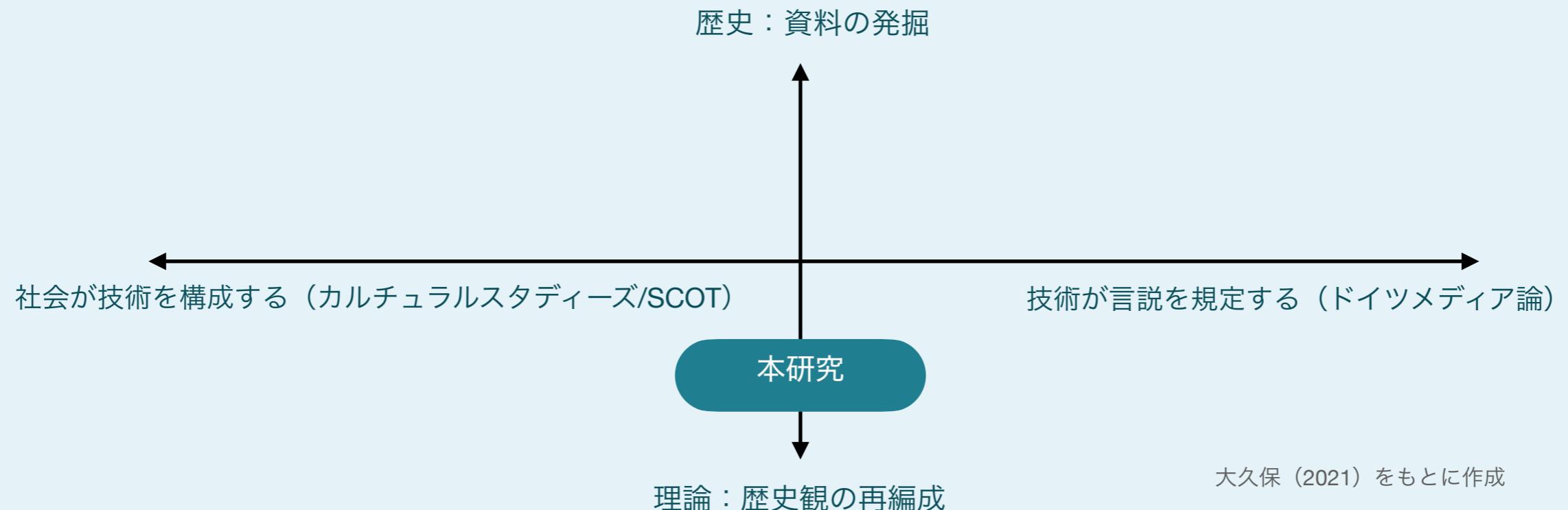
- 音楽のためのプログラミング言語制作は汎用言語に比べて圧倒的に設計を試みる人が少ない

イントロダクション

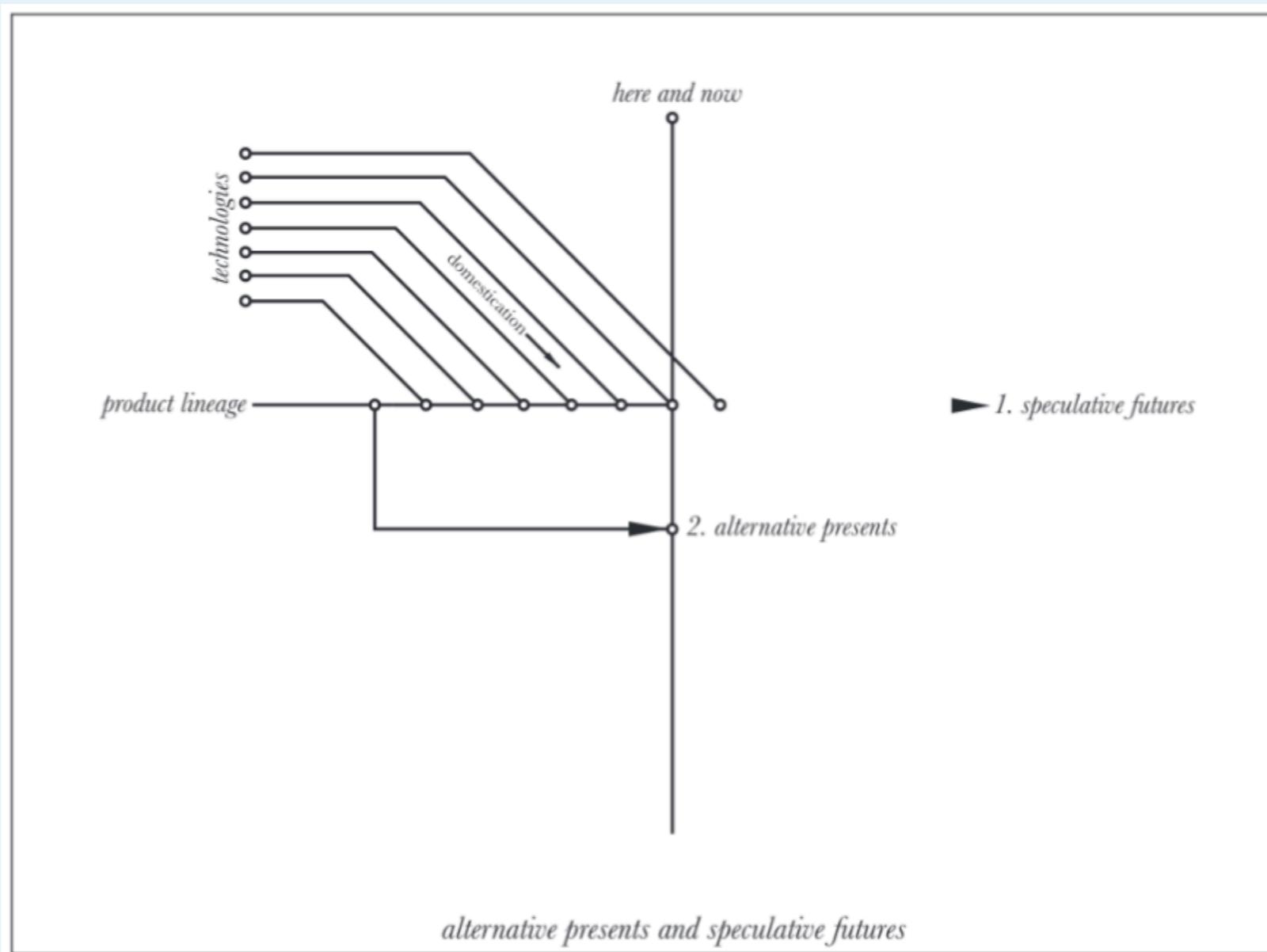
方法論



メディア考古学



- 歴史の中に埋没したメディアを掘り返すことで所与のものとされている歴史に別の物語を与えるアプローチ（フータモ, 2014）
 - 本研究では現在のコンピューター環境を、**誰もが自由にプログラミングできるメタメディアとしてのコンピューター**という理想が不完全な形で実現され、忘れ去られたものとして捉え直す
 - エマーソンによって詩と視覚表現における同様の議論がなされている（Emerson, 2014）

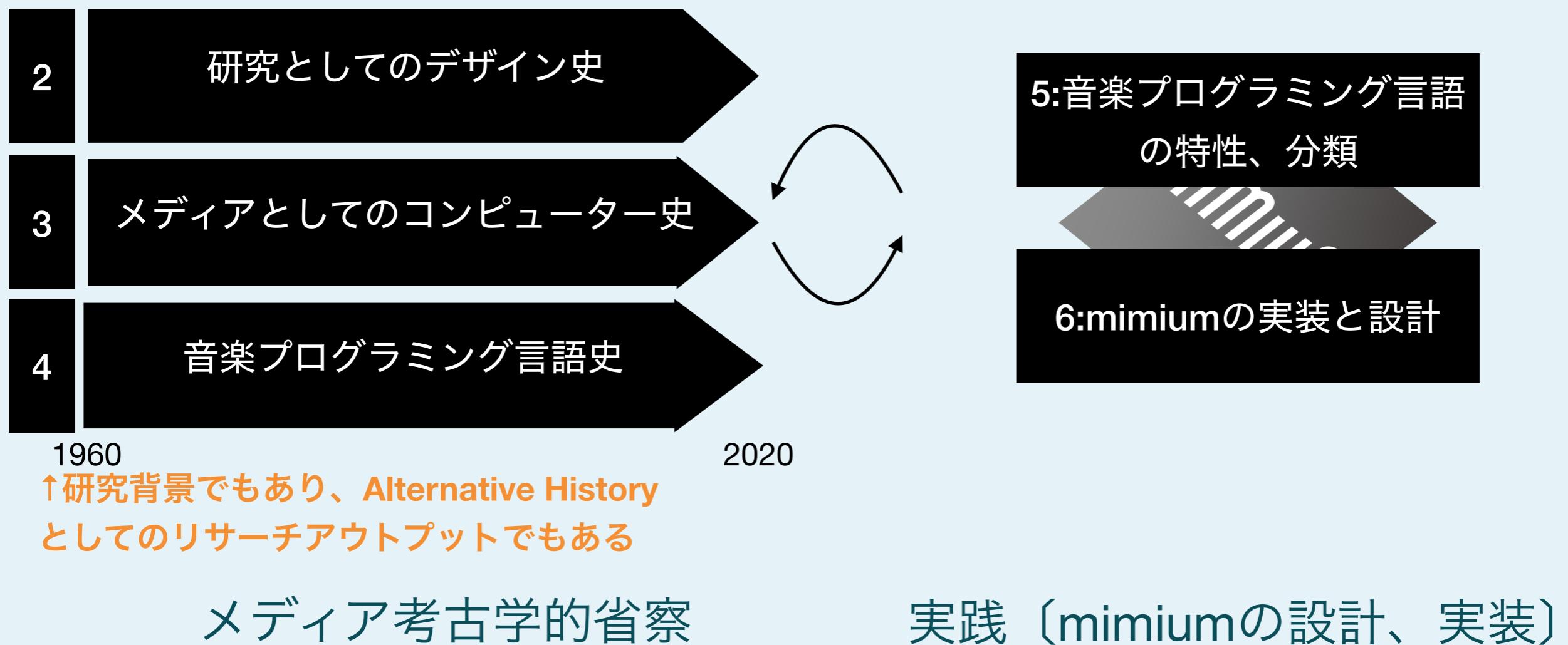


James Auger, Alternative Presents and Speculative Futures (2010)

メディア考古学は、過去の技術のリサーチからありえたかもしれない現在/ありうるかもしれない未来の技術の姿を想像する、クリティカル・デザインの概念とも共通している

イントロダクション

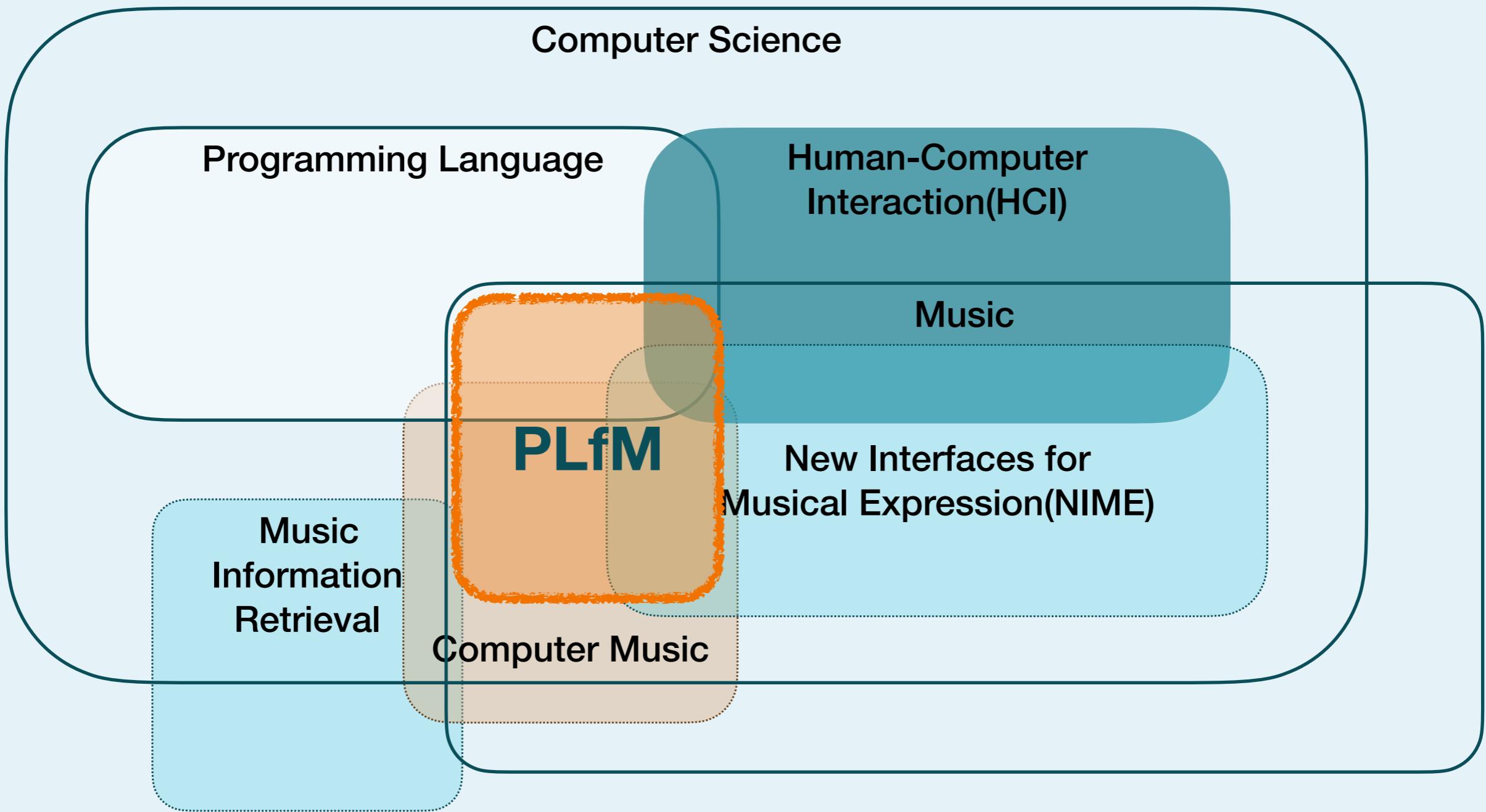
章構成



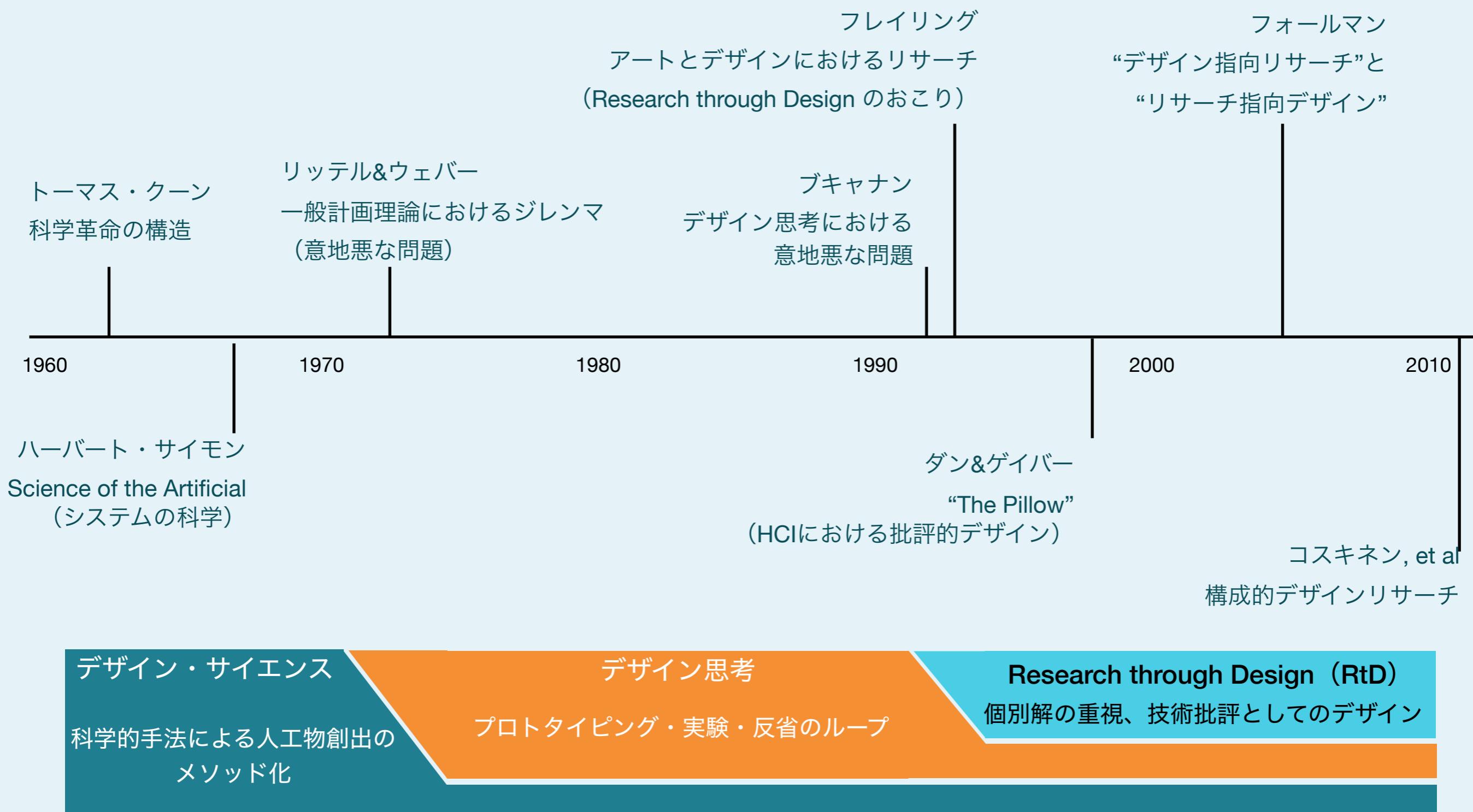
2: How: PLfM研究の位置付け

隣接領域における研究方法論の議論

↓Computer “Science” だけど、自然科学よりもデザイン学の中に位置付ける動きがある



デザイン学としてのHCIの系譜



HCIと批評としてのデザインの接続

デザインは科学的理論と学問を基盤にするべきだ。

システム（人工物）は現実の、共有された問題に働きかけるべきだ。

システムは曖昧さのない解を提供すべきだ。

共感やインスピレーションは日常生活をデザインする上で不可欠だ。

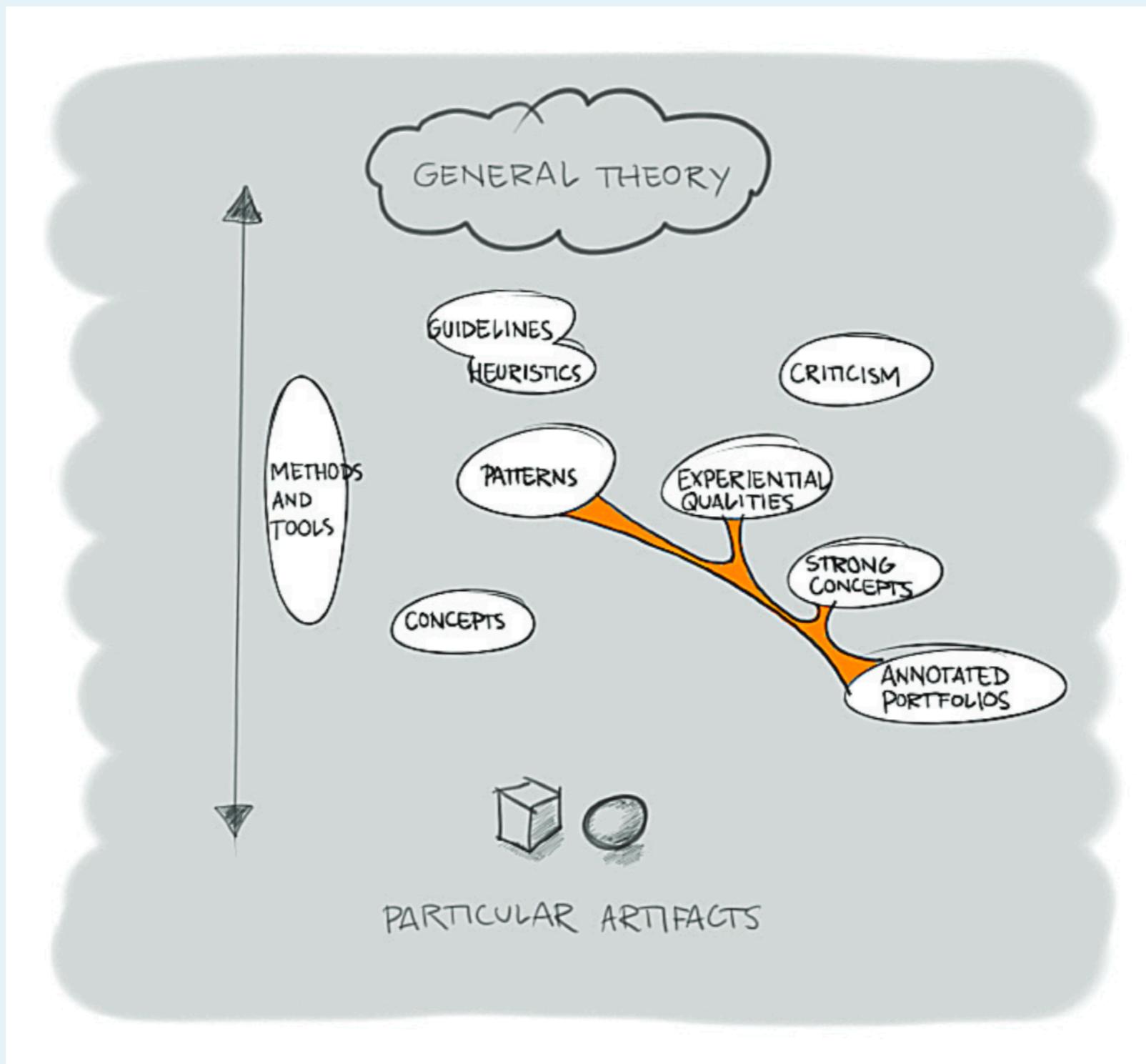
テクノロジーは奇特性や活動や取り組みへの探求を誘発できる。

曖昧さは想像力と洞察を誘発するためのパワフルな道具だ。

Subjective Design for Everyday Life, Gaver, Boucher, Pennington & Walker(2003)

...とはいって、プログラミング言語のような、ある程度汎用性のあるツールに関しても同様の議論ができるだろうか？

「中間的な知」の創出としてのデザイン



Jonas Löwgren “Annotated Portfolios and Other Forms of Intermediate-Level Knowledge”(2013)

プログラミング言語の評価の議論

- プログラミング言語研究はHCIよりは科学（というか論理学）的手法に基づく研究が多い
- しかし、プログラミング言語における「直感的」、「効率的」、「汎用的」などの意味がはっきりしない評価用語たちの存在は近年問題になってきている
- プログラミング言語研究には**主張と根拠の整合性(Claim-Evidence Correspondence)**が不足 (Markstrum, 2010)
 - ベンチマークや評価実験といった量的で再現可能な根拠があっても、それが正しく主張（効率よくプログラムが書ける）に結びついてないものがある
- 実行速度のような量的指標と効率的というユーザーベンチマークという質的結果の関係（量→質の転化）
- メモリ空間-計算時間のような量的、動的変更-実行最適化のような質-量間の、根本的トレードオフがある中での選択の理由をどう正当化するか

デザインリサーチ的PLfM研究

- 例えば西野による、microsound (Roads,2003) を記述しやすくするためのLC言語の開発をRtDの文脈に位置付けた研究(Nishino, 2012)
 - しかし、単なる問題解決に陥っている感も否めない
 - 研究を通じた問題発見、自らの視点の変化というReflectiveな側面は？
- 音楽プログラミング言語デザインを通じて、オルタナティブな音楽文化のメタ・デザイン、あるいは概念の創出を行ってきた研究こそRtDの文脈におけるもの
 - Alex Mclean TidalCycles(2014)とAlgorave
 - Andrew Sorensen Impromptu/Extempore(2010~)とCyber-physical Programming
 - Thor Magnusson ixiQuarksとEpistemic Tool概念(2009)

2:音楽プログラミング言語研究の位置付け

マグヌッソンの認識論的道具としてのDMI

- ・マグヌッソンはコンピューターを用いた楽器（Digital Musical Instrument:DMI）がアコースティック楽器とどう異なるかを、技術哲学や社会科学技術論（STS）の理論をもとに人文学的に考察
- ・自らも、ixiQuarksというオーディオビジュアルライブコーディング環境を開発
- ・その他言語開発者へのインタビューも交え、DMIの特徴として認識論的道具（Epistemic Tool）という概念を提示

2:音楽プログラミング言語研究の位置付け

マグヌッソンの認識論的道具としてのDMI

- ギブソンによるアフォーダンス：「環境に埋め込まれた人が行動できる可能性」
- DMIとアフォーダンス (Tanaka 2010)
 - 楽器の持つ演奏者の音楽的表現を引き出すアフォーダンス
 - それを見る/聞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
 - DMI演奏を楽器-演奏者-聴衆の三者による社会的行動（「参加する音楽」のひとつ）として捉え直した
- マグヌッソンはこれをさらに、時間方向に拡張
 - 誰かが作った楽器を改良したり、それをもとに新しい楽器を作ったり...という、何年もかけて発展していく音楽文化としてDMIを特徴づける

“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。”

Thor Magnusson “On Epistemic Tools”(2009)

- 狹義的には、アコースティック楽器や、アナログ電子回路のように、エラーや製作者の予想しない演奏法が起きる余地が少ない
- 想像しうるどんな音でも作れる↔想像できない音は絶対に作れない

2:音楽プログラミング言語研究の位置付け

ユーザー中心主義と認識論的道具

- ・ デザイン学にアフォーダンス概念を持ち込んだ認知心理学者のドナルド・ノーマンはあまり着目されてないが、必ずしも良い技術がそのまま社会に受け入れられるわけではないこと、特に、**インフラストラクチャ**の力によって変化に時間がかかるってしまうことを念頭に置いていた
- ・ ノーマンは情報アプライアンス（コンピューターを使った電子機器）の良い例として、MIDI（電子楽器の共通規格）が電子楽器に成功をもたらしたと挙げている
- ・ しかし、MIDIこそ西洋音楽文化の様式を埋め込んだプロトコルであり、MIDIを使った楽器は認識論的道具の代表とも言える
- ・ Open Sound Control(Wright, 1997)のようなより包括的なプロトコルが作られても、MIDIを置き換えることはなかった（ノーマン自身が言った、「古いやり方がしぶとく残る」状況）

2:音楽プログラミング言語研究の位置付け

時間のかかるもののデザイン

- ・ノーマンは2010年以降、複雑なものを隠すことで認知的負荷を下げる（インビジブルコンピューター）という思想から、本質的に複雑なものは単純化しうが無いと立場を変えるようになる
- ・Design X (2014)では、変化に時間がかかる問題、利害が人によって異なる社会問題全般の解決に向けたマニフェスト
 - ・→時間がかかるとしても、インフラストラクチャ自体を設計することが重要だという考えにも読める
 - ・カーネギーメロン大のTransition Designなど近年のRtDの動向にも共通
- ・またプログラミング言語にはそもそも明確な完成の概念がない、メジャーな言語でも開発からリリースまで3年以上掛かることが普通にある (Faustなど)

2:PLfMの学術的位置付け

音楽のためのプログラミング言語の研究は...

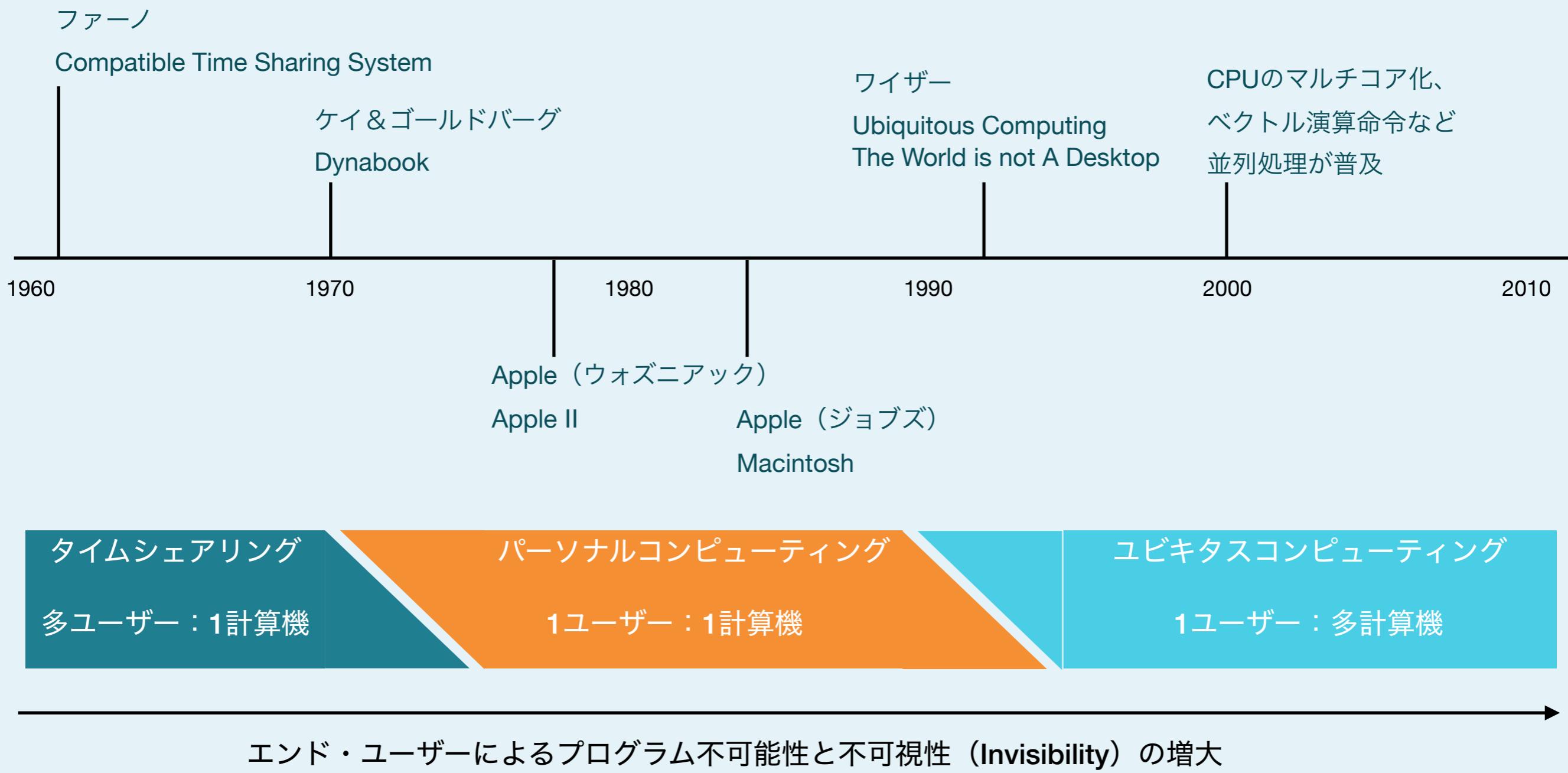
完成したものを対象として議論しなくともよい。

しかし、完成を目指して実用しながら検討していればなおよい。

客観的な証拠だけがその言語の学術的価値を裏付けるものではない。

言語を作る経験を通した発見の共有も、デザイン研究としての知のあり方である。

3. 音楽家の実践としてのPLfM設計 メタメディアとしてのコンピュータ再考



3. 音楽家の実践としてのPLfM設計 メタメディアとしてのコンピュータ再考

- ユーザーによるカスタマイズの2段階(Lieberman & et.al, 2016)
 1. パラメーター化でユーザーに選択肢
 2. プログラム 자체をユーザーが変更可
- アラン・ケイとアデル・ゴールドバーグが提示したPCの姿には、Smalltalkというプログラミング言語で自らの道具を自らが作ることが重要視されていた



コンピューターのブラックボックス化

- Apple Macintosh 家電としてのコンピューター
 - Dynabookに強く影響を受けたにも関わらず、で複雑な拡張機能を隠蔽することで「ユーザーフレンドリー」な機器に
 - マーク・ワイザー “The World is not a Desktop”
 - コンピューターをKnowledge Navigatorではなく、人間の能力を拡張する環境として、「ユビキタス・コンピューティング」を提起
 - ユーザーフレンドリーの延長として、良い道具とは人に意識されない「Invisible」なものであることを強調



初代Macintoshの背面 (CC-BY-SA, <http://www.allaboutapple.com/>)

音楽家のコンピューターに対する向き合い方

- 提供された音楽制作のためのソフトウェアを利用する
 - →限られた道具の拡張性
- 自らソフトウェアとして音の作品を制作する
 - →プラットフォーム(App Store)に作品の形式が規定されてしまう
- 道具を想定されない方法で用いる（ベンディング的）
 - →失敗することも許されない（スタイルへの回収/そもそも誤用できない）

ソフトウェアとしての音楽の失敗

- Audible Realities 徳井直生らの音楽以前だが音の聞き方の変容を促すようなiPhoneアプリ群
- 2018年、「死んだ」作品の展示を行う「メディアアートの輪廻転生」に出品
 - シンプルすぎてApp Storeの審査に通らない



マンス

内課金

スクリプション

他の購入方法

リーダー」App

マルチプラットフォームサービ

ンタープライズサービス

個人対個人のサービス

Appの外部で使用する商品

料のスタンドアロンApp

のハードウェアを必要とする

通貨

le Pay

のビジネスモデルの問題

される行為

されない行為

的に機能し、新規および既存ユーザーの双方にとって魅力的であり続けるよう、アップデートしていく必要があります。

機能しなくなったり、サービスの品質が低下したりしたAppは、隨時App Storeから削除される可能性があります。

4.1 模倣

独自のアイデアを生み出してください。自分だけのアイデアがあるはずです。それを形にしましょう。App Storeで人気のある最新Appを単にコピーしたり、別のAppの名前やUIを少し変更して自分のAppとして提出したりすることは避けてください。知的財産侵害の申し立てをされるリスクがあるだけでなく、App Storeの運営が困難になり、また、他のデベロッパにとってもフェアではありません。

4.2 最低限の機能性

Appを作成する際は、Webサイトを単に再パッケージしたようなものではなく、優れた機能、コンテンツ、UIを作成するようにしてください。特に便利でも、ユニークでも、「Appらしく」もない場合、そのAppをApp Storeで提供することはできません。Appが継続的に楽しめる何らかの価値、または十分な有用性を備えていない場合は、承認されない可能性があります。Appが単に曲または映画の場合は、iTunes Storeに提出してください。Appが単に書籍またはゲームの攻略本の場合は、Apple Books Storeに提出してください。

4.2.1 ARKitを使用するAppは、表現に富み、統合された拡張現実体験をユーザーに提供する必要があります。単にモデルをARビューで表示したり、アニメーションを再生したりするだけでは不十分です。

4.2.2 カタログを除き、主な目的がマーケティングの資料の提供、広告、Webクリッピング、コンテンツアグリゲーター、リンク集であるAppは許可されません。

4.2.3

- (i) Appは独立して機能するものである必要があります。そのAppを実行するために別のAppのインストールを求める場合は、

• App Store 審査ガイドライン <https://developer.apple.com/jp/app-store/review/guidelines/>

• 想像できない音は絶対に作れない→想像できたとしても作れない

ベンディング的アプローチの限界

誤用対象へのアクセス不可能性

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。

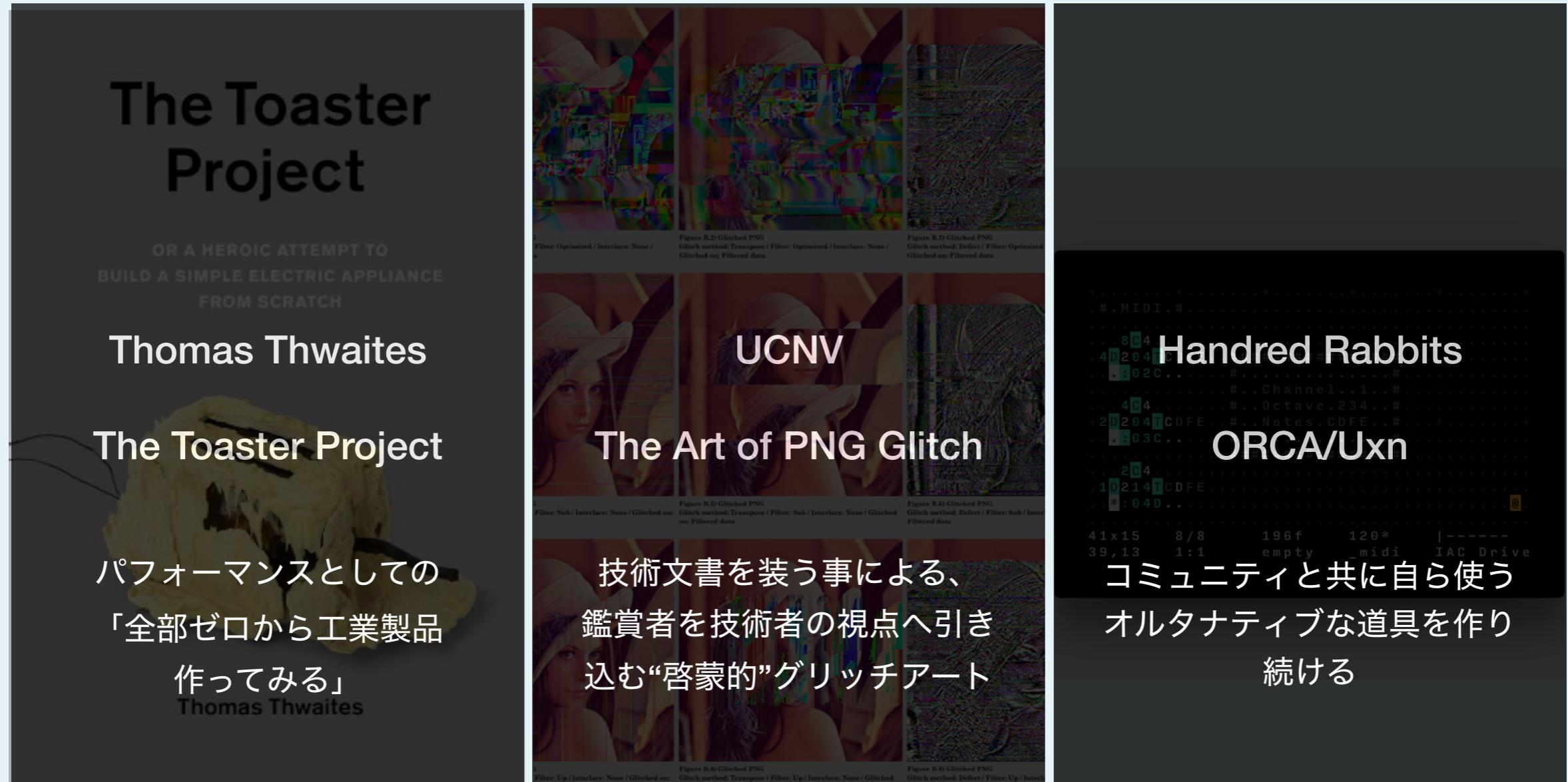
今日のおもちゃは单一の邪惡な黒い物体に全てがコントロールされていて、再配線可能な接続箇所が1つも見つからないことがあります。（中略）今やサーキットベンダーたちは改造可能な中古オモチャを求めて、リサイクルショップやeBayを物色してまわる運命です。（コリンズ、2013、強調は筆者）

ベンディング的アプローチの限界

しかし、その音があたかも故障や誤作動から生じたように聞こえることから、グリッヂと呼ばれていたこれらの試みは、イマン・モラディがPure-Glitch（自然発生的なデジタルのエラー）とGlitch-alike（本物のグリッヂを模したデジタルの人工物）を対比したように（Moradi 2004）、発生環境としてのデジタルの特性をあらわにするという当初の実験から、プリセットやプラグ・インないしはあらかじめ用意されたサンプルを使ったその結果の模倣へと姿を変化させていった。

いみじくも〔かつて「失敗の美学」としてグリッヂのようなアプローチを美学的に正当化した〕カスコーンが二〇十四年に記したエッセイで、「グリッヂはiTunesのジャンルのタグになった」（Cascone 2014）と述べたように、もはやデジタル技術の失敗から音を生み出そうとした「ポスト-デジタル」はソフトウェアの通常の「機能」としての「前景」になってしまった。（城、2021、強調は筆者）

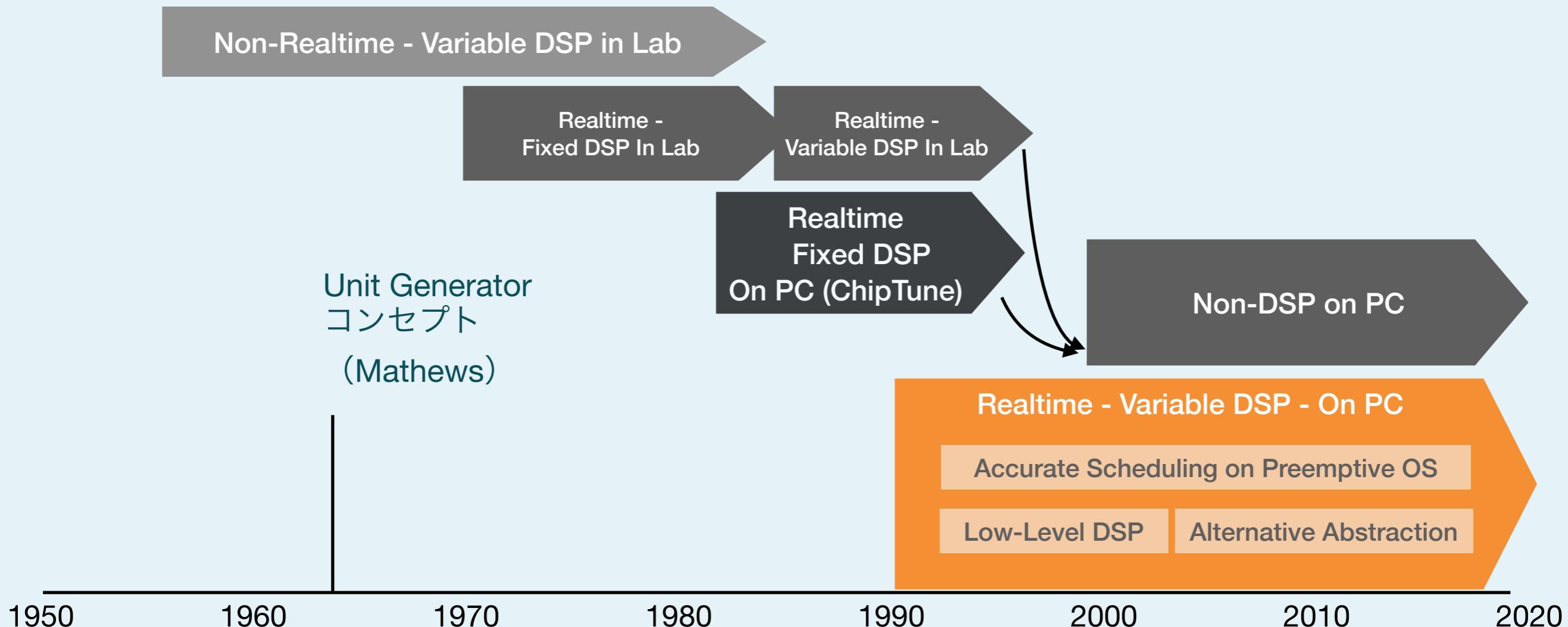
ブラックボックスを開く行動



不可視化されているなテクノロジーを自ら切り開き、オルタナティブな環境を構築することは、デザイン・アクティビズムでもあり、今日技術を積極的に用いるアーティストが可能な数少ないアプローチ

4. PLfMの歴史

- Realtime : リアルタイムに音が生成できるか?
- Variable DSP: オシレーターやフィルターのようなモジュールをソフトウェアで (理論上) 無制限に作れるか?
 - Non-DSP: MIDIなど、音価レベルの処理に集中したもの (MML、SuperColliderクライアント系など)
- Lab or PC: 大学の研究所などで行われているか、パーソナルコンピューターで使えるか



どこからが積極的手段としての言語？

“コンピューター言語とはユーザーに、そのプログラムにおける問題の領域と関係のない細部のことに気を使わずにプログラムを書かせられるような、計算の抽象モデルをあらわす。”

“コンピューター音楽言語は作曲や信号処理のアイデアを可能な限り簡単かつ直接的に表現できるような抽象化の集合を提供するべきだ。”

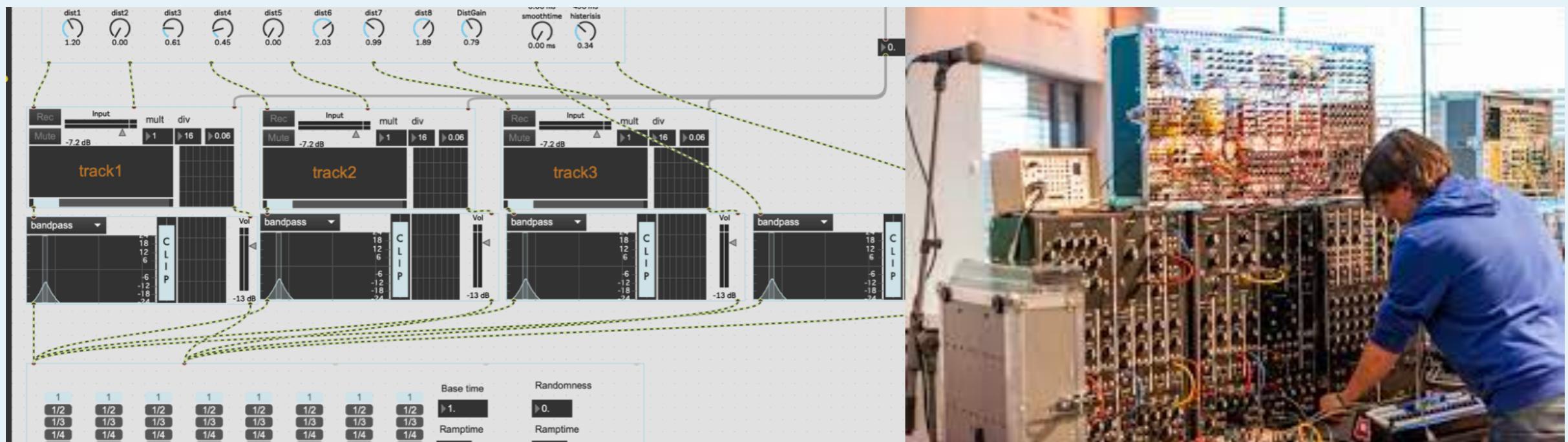
(McCartney, 2005)

- Dynabook(1970)のような対話的GUI以前はパンチカードやテキストしか入力インターフェースがなかった→消極的な意味での「言語」という選択肢
- 1980年代IRCAMでは音楽のための言語 (Chant/Formes) を使うためにFORTRANやLISPといった下層の言語やハードウェアの知識が依然必要だった(Born, 1995)
- →90年代以後の、ハードウェア的制約のない、かつGUIでの操作も可能な中で敢えて言語というインターフェースを選択している言語がより積極的な意味でのPLfMと言える

ただし逆説的に音楽家にCやC++などの汎用言語を遠ざけ分業を深めた要因ともいえる

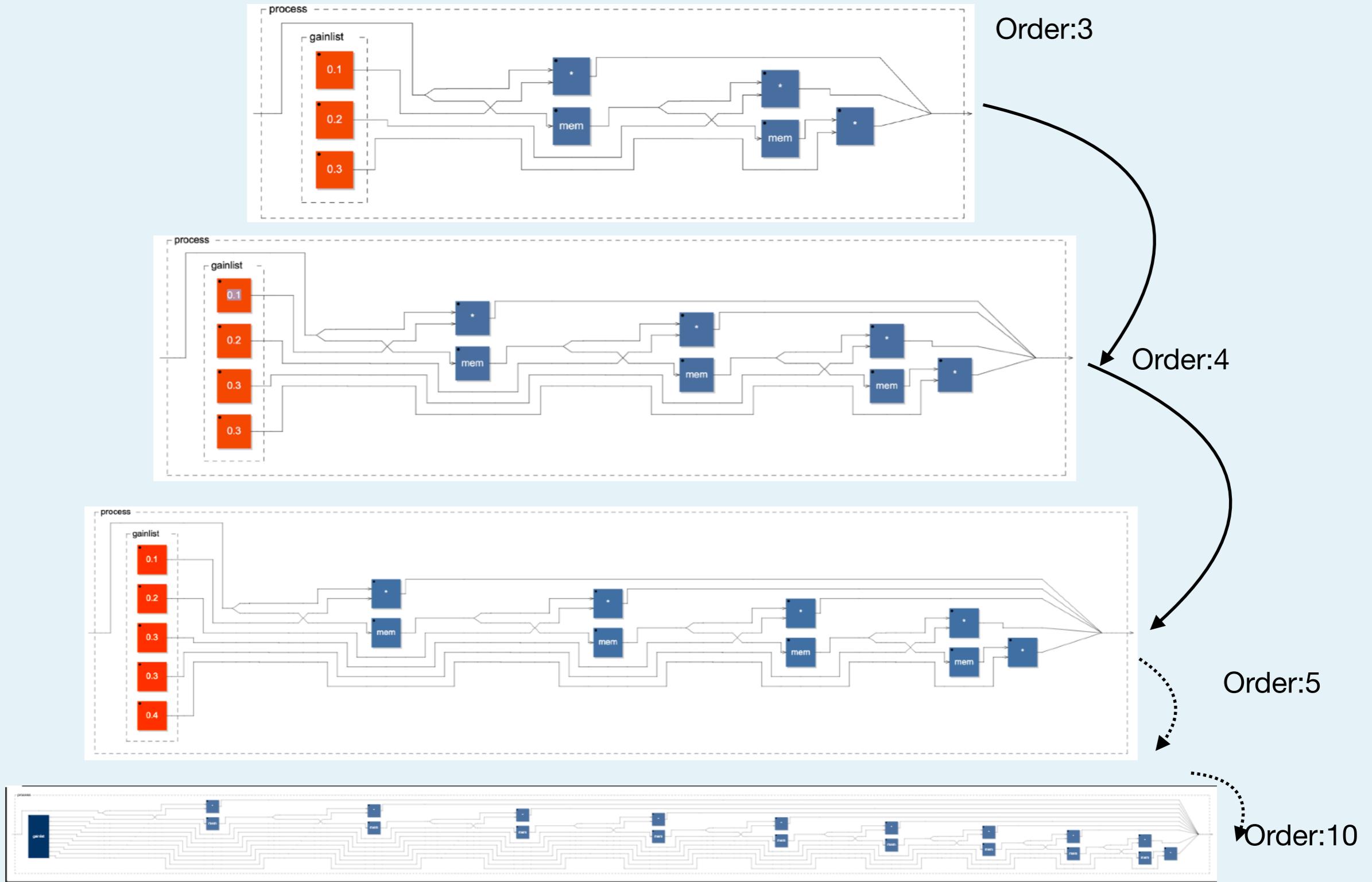
Unit Generator(UGen)コンセプト

Maxはスクリーン上でパッチングする言語を作るという、パッチ可能なアナログシンセサイザーのモダリティを模倣する試みだった。(Puckette, 1997)



Unit Generatorという概念そのものはモジュラーシンセサイザーと同時期に発生したもの (Mathews 1963; Mathews and Roads, 1980) だが、PuredataやMax、SuperColliderなど90年台以後の言語は明らかにモジュラーシンセサイザーの模倣として生まれてきている=MIDIほどではないにせよ、ある音楽様式を埋め込んでしまっている

UGenはその言語上では操作不能なブラックボックス
(オープンソースだとしてもCなどの汎用言語で実装しなければならない)



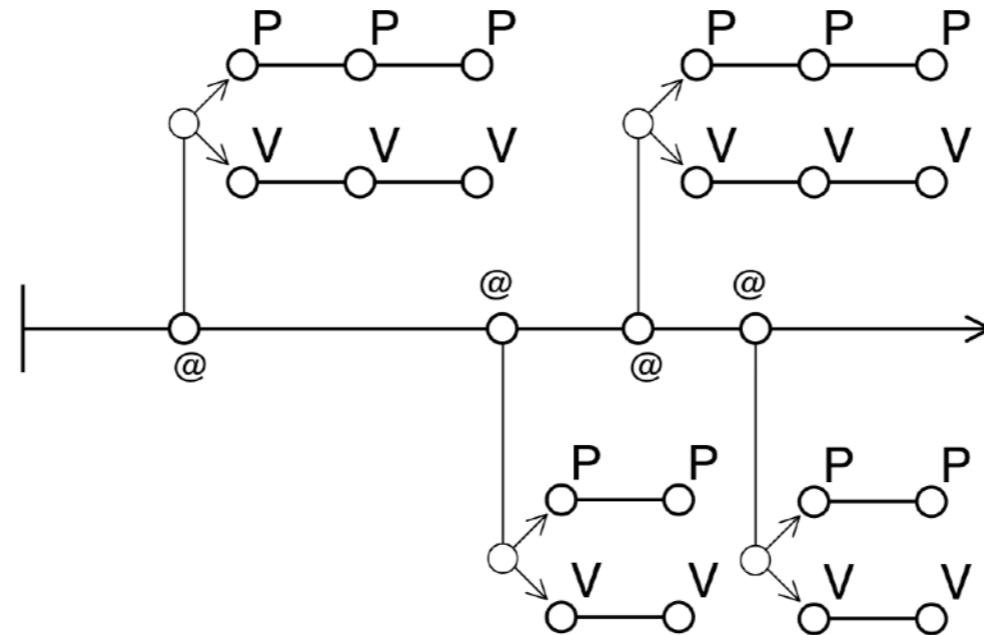
例えば、Unit Generatorコンセプトではこのような接続方法そのものの抽象化が大変難しい
(画像はFaust言語による例)

「いつ」計算するのか問題

2000年代以後の計算インフラストラクチャの問題

- 2000年代以後、CPU自体が複数のコアを持ったり、1つのCPUが1クロックで同時に複数の数値の足し算や掛け算などを可能にする命令（SIMD）を持つように
- さらに、オペレーティングシステム（OS）が複数のアプリケーションの実行を時分割＆各コアに割り振っており、厳密にいつ処理を行うかをユーザーが指定できない
 - OSの機能を用いて別のスレッドで並行処理を指定したとして、**本当に別のコアで同時に計算してるか、同じコア上で仮想的に時分割されてるかが不可知**
- また音声データを厳密に実時間で処理せず、64や256サンプルなど、適当な単位に区切ってまとめて処理し、計算の順序だけを保証するように（論理時間）
- しかし、鍵盤からの入力など、実時間との整合性もとる必要がある
 - イベントを一度蓄積し、オーディオのまとまった計算の前後でまとめて消化する、並行して別のスレッドで定期的（20msごととか）に消化する、イベントを論理時間上のイベントとして一旦紐付ける、など言語ごとに対応がかなり異なる

計算モデルの抽象化の試み



(Pitch vec \times Pressure vec) event ivec

- Chronic(2002) : Brandtによる、音楽や映像のような時間の関わる表現のデータを、
 - Event (ある時間上に配置されたデータ)
 - Vector(有限長の配列)
 - Infinite Vector(無限に配列データを生成し続ける関数)

の代数的組み合わせで表現しようとした試み（しかし、リアルタイムに動作できなかった）

最低限の抽象化=並行性も含め、時間とともに発生する計算を統一的に扱える抽象計算モデルは未だ存在しない
(例えば、汎用言語ではラムダ計算のようなハードウェアに依らない抽象化が活用できている)

2000年以後の主要な焦点

Multi-Languageパラダイム

1.正確なイベントスケジューリング：**ChucK**、**LC**、**Extempore**等
プリエンプティブマルチタスク（ユーザプログラムがOSのタスクスケジューリングに
関与できない）環境下でどうにか正確なイベント処理を目指す

2.低レイヤの形式的抽象化：**Faust**、**Soul**、**Kronos**、**Vult**
信号処理のプリミティブ(UGen)はブラックボックスになっておりC++で記述
信号処理とイベント処理で別の型の同じ処理(+と+~とか)を使い分けなきゃいけない

3.高レイヤのUGen以外の形での抽象化：**Sonic Pi**、**TidalCycles**等
言語そのものの拡張性、自己反映性に限度がある
特定の音楽領域にさらに特化した言語/ライブラリを作りたい

可能な表現を局所化することで、各々の需要に応じた「最低限の抽象化」を生み出してきている

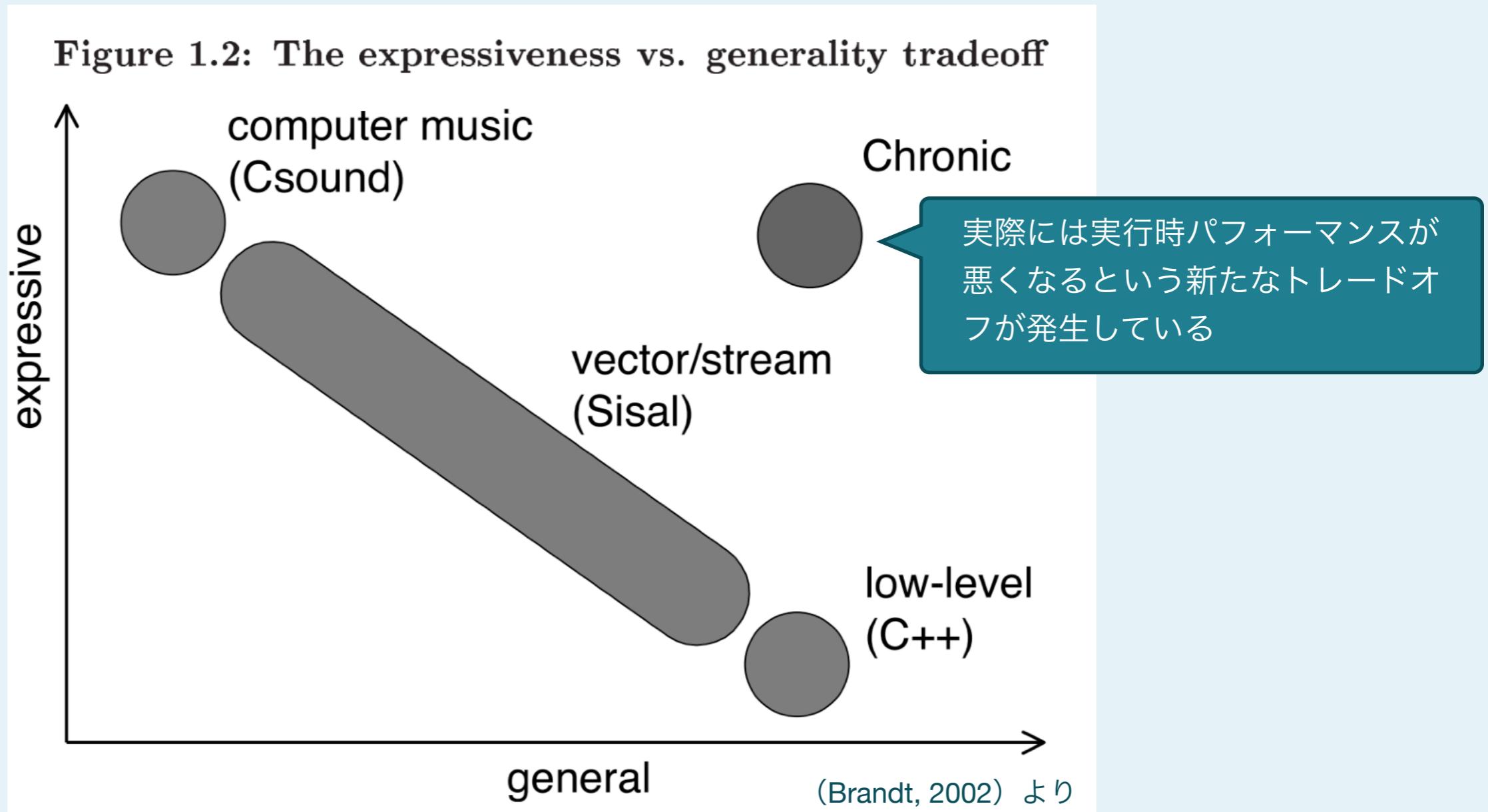
4. PLfMの歴史

- 音楽のためのプログラミング言語の歴史は、プログラマー（＝ユーザー）音楽に関連するタスクに注力させるべく“最低限の抽象化されたモジュール”的提供を目指してきた。
- 現状、ユーザープログラムが関与できない汎用OSのタスクスケジューリングという、インフラストラクチャへの依存をどうにか切り抜ける必要がある。
- ある時間に対応して起きる（しかも複数タスクが並行しているかもしれない）計算を表現できる決定的な抽象計算モデルは無い。常に実装依存
- 結果的にMulti-Language パラダイムという各表現のレイヤーごとに、言語の抽象化方法を切り分け、**互いの存在をInvisibleなもの**としてきたという、3章で考察したメディアとしてのコンピューター史とも対応する

5. PLfMの特徴や分類

- 実装方法や、特徴、評価の語彙を汎用言語の議論も参照しつつ、一般化を試みる
- 汎用言語でも、計算時間-メモリ空間トレードオフがあるように、PLfMの設計にもトレードオフがあることが示せれば、設計方針の主張と根拠の整合性をはっきりさせることができる
- コンピューター音楽言語を特徴付ける技術的要素はすでに整理されている(Dannenberg, 2018)
 - シンタックス（統語論） - 言語の表面的見た目を特徴づけるもの
 - セマンティクス（意味論） - 言語が実行されるときの意味を決めるもの
 - ランタイム - その言語で書かれたプログラムを実際に動作させるためのホストになるプログラム
 - ライブラリ - その言語で記述された、よく使われる機能を集めたもの
 - 開発環境（IDE） - その言語を書くためのエディタやデバッグのためのアプリケーション
 - コミュニティやドキュメントなど

主張に用いられる言葉とトレードオフ



- ここでExpressivenessとは“読みやすく書きやすい”、Generalとは“可能な表現の範囲が広い”という意味だが、Expressiveを可能な表現の範囲が広いとして使うものもある (Kronos (Norilo, 2015) など)

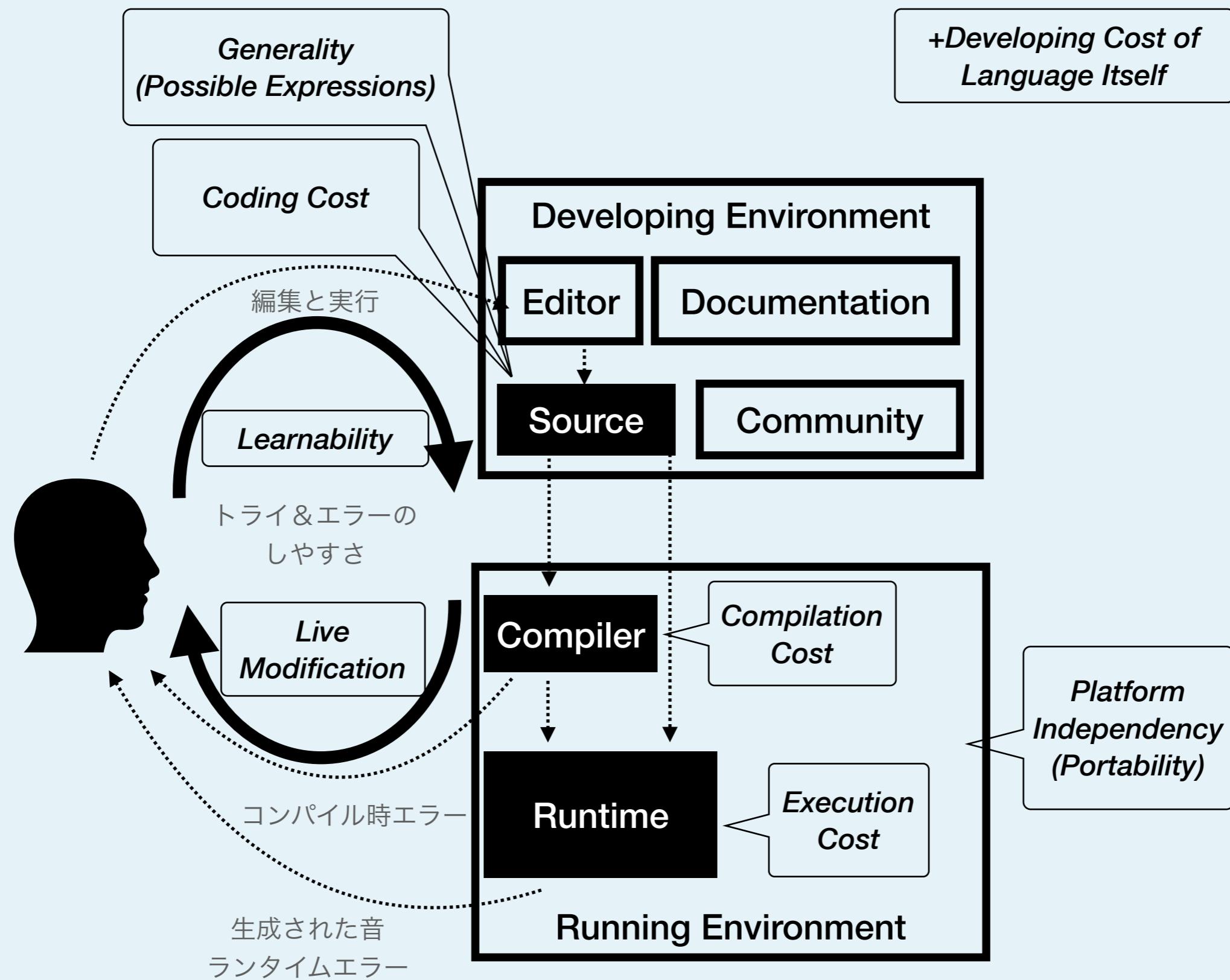
主張に用いられる言葉とトレードオフ

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。

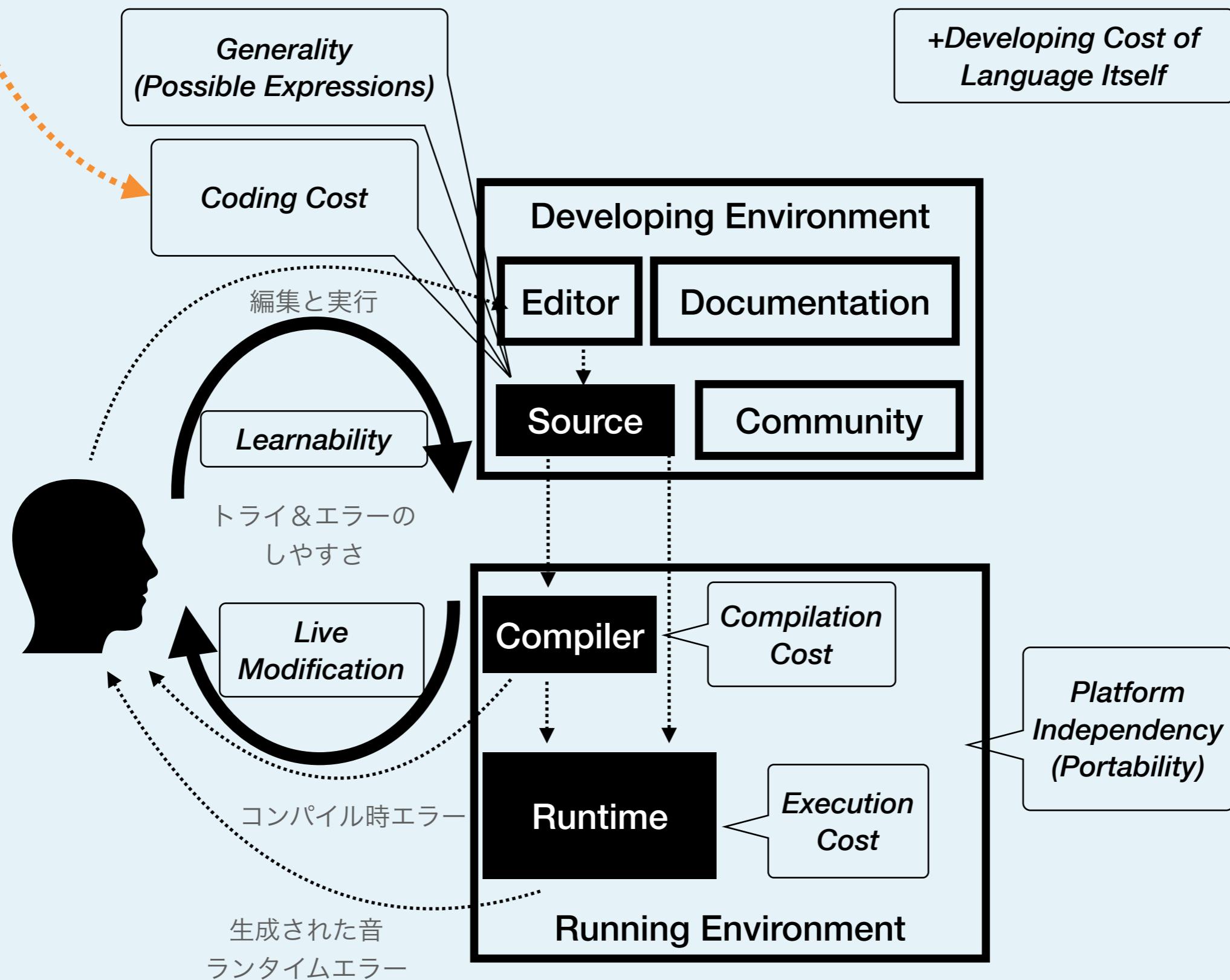
タスクに応じてプログラマは異なる複雑度のエントリーポイントを選択することが可能になる。より低く汎用的なレベルではより複雑なコードの設計が必要だが幅広い結果を得られる。一方、より高い、特殊化、具体化されたレベルではプログラミングに必要な労力という観点からプロセスはより効率的になるだろう。 (Lazzarini, 2013,強調は筆者)

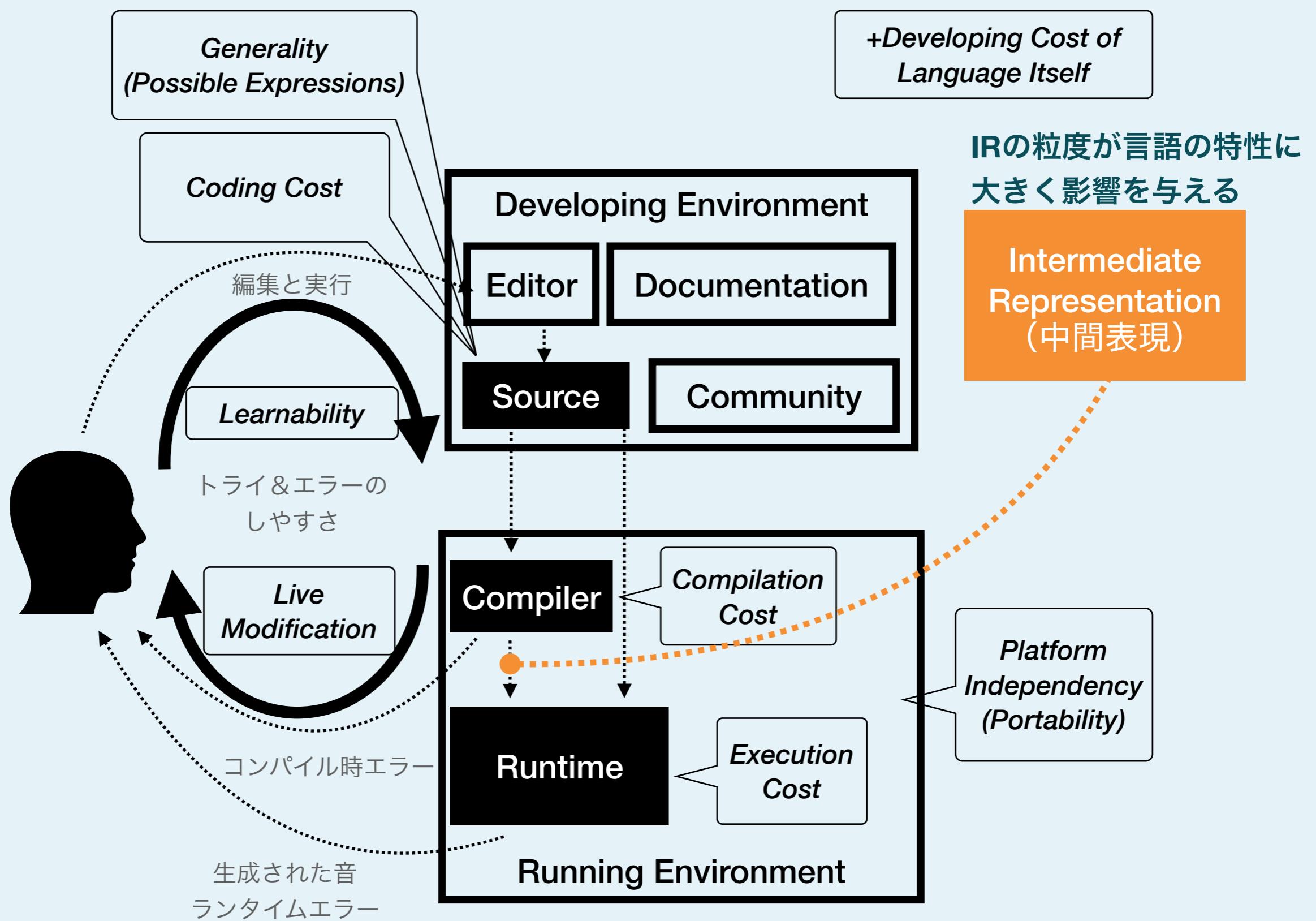
- ここでのEfficiencyとはユーザーのプログラミング体験に関わる意味合いだが、この言葉は一般に計算速度の速さを意味することもあり、誤解を招きやすい

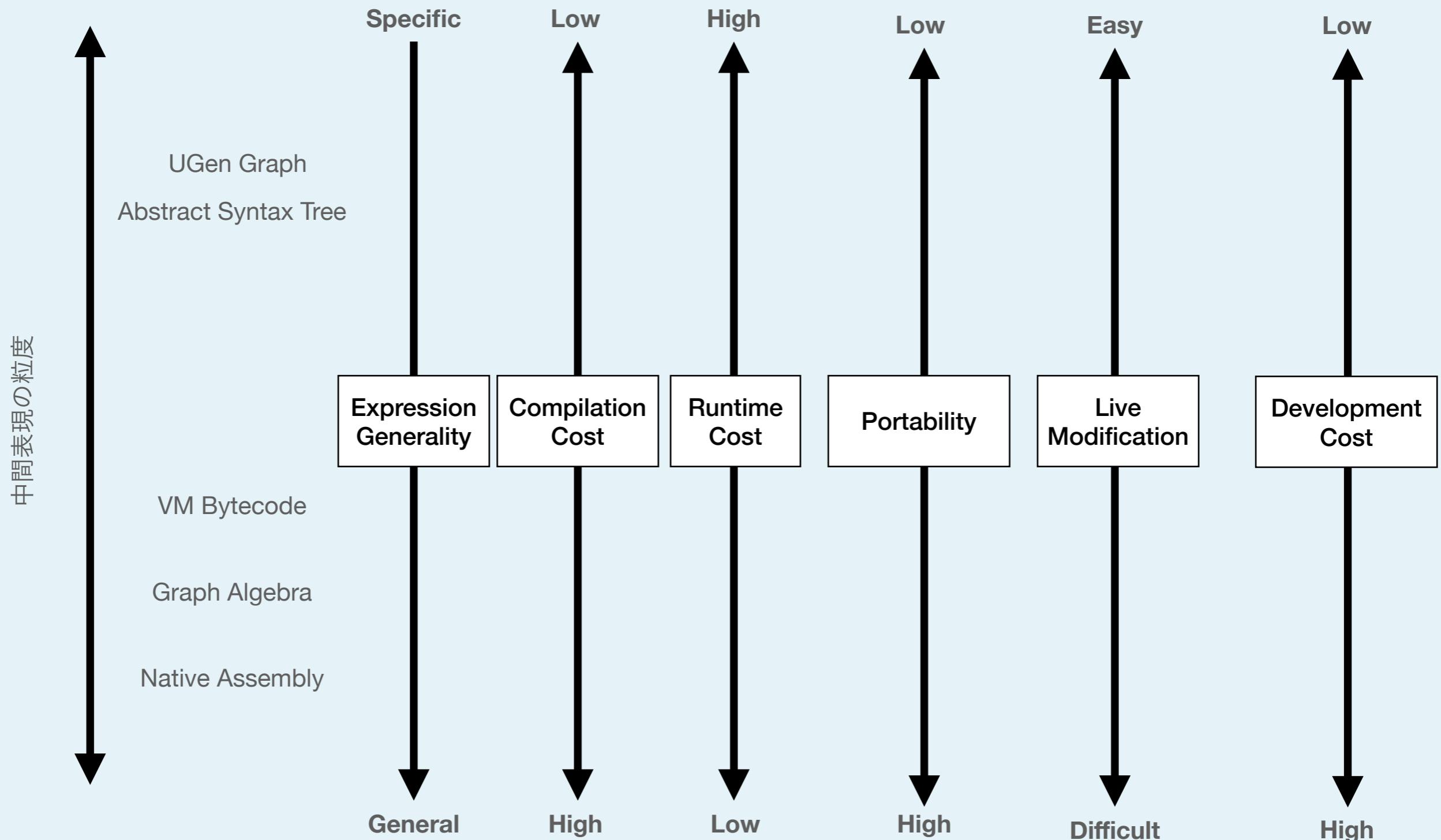
一般化した実行モデル



BrandtのいうExpressivenessとLazzariniのいうEfficiencyはここのこと



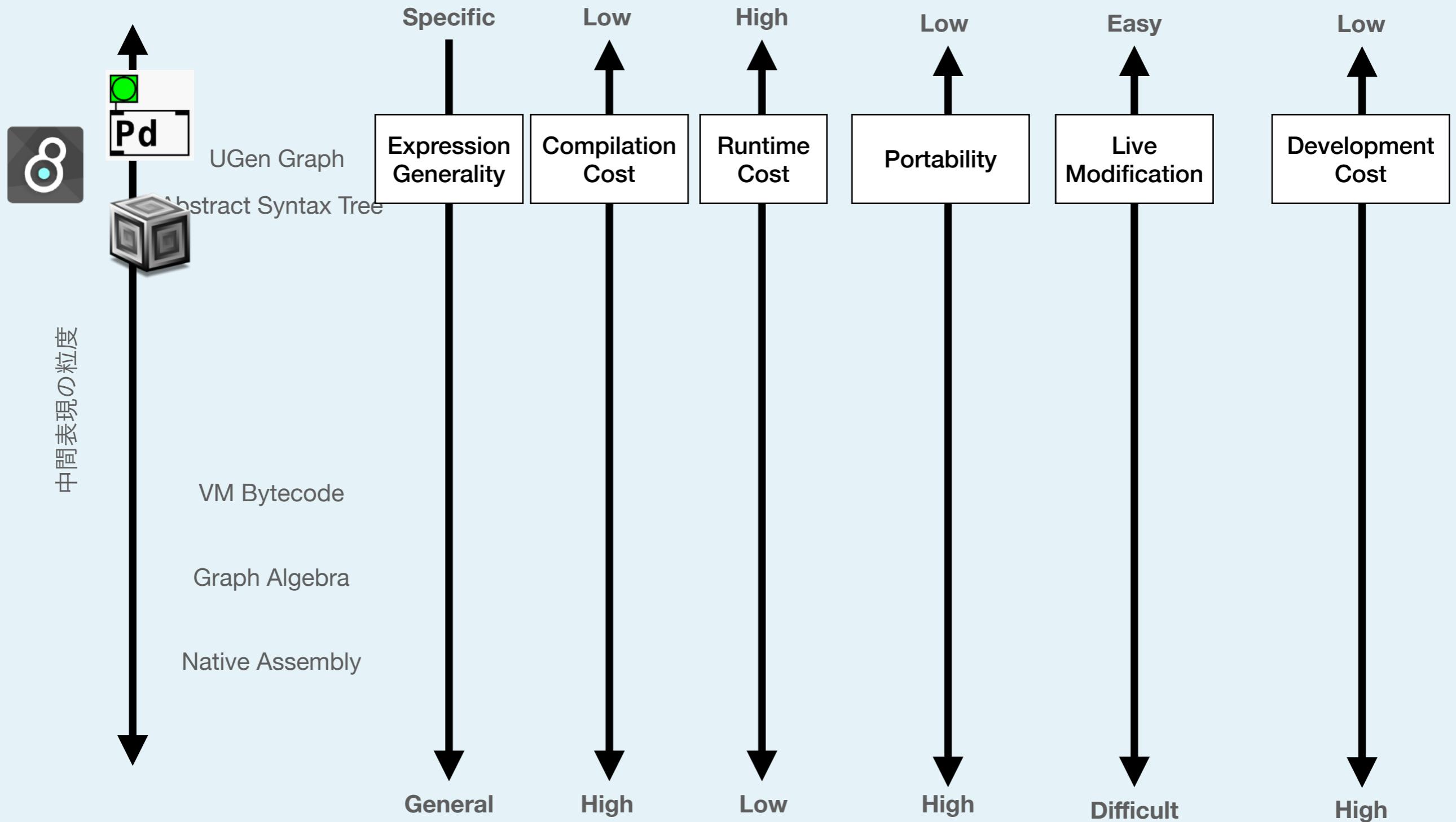




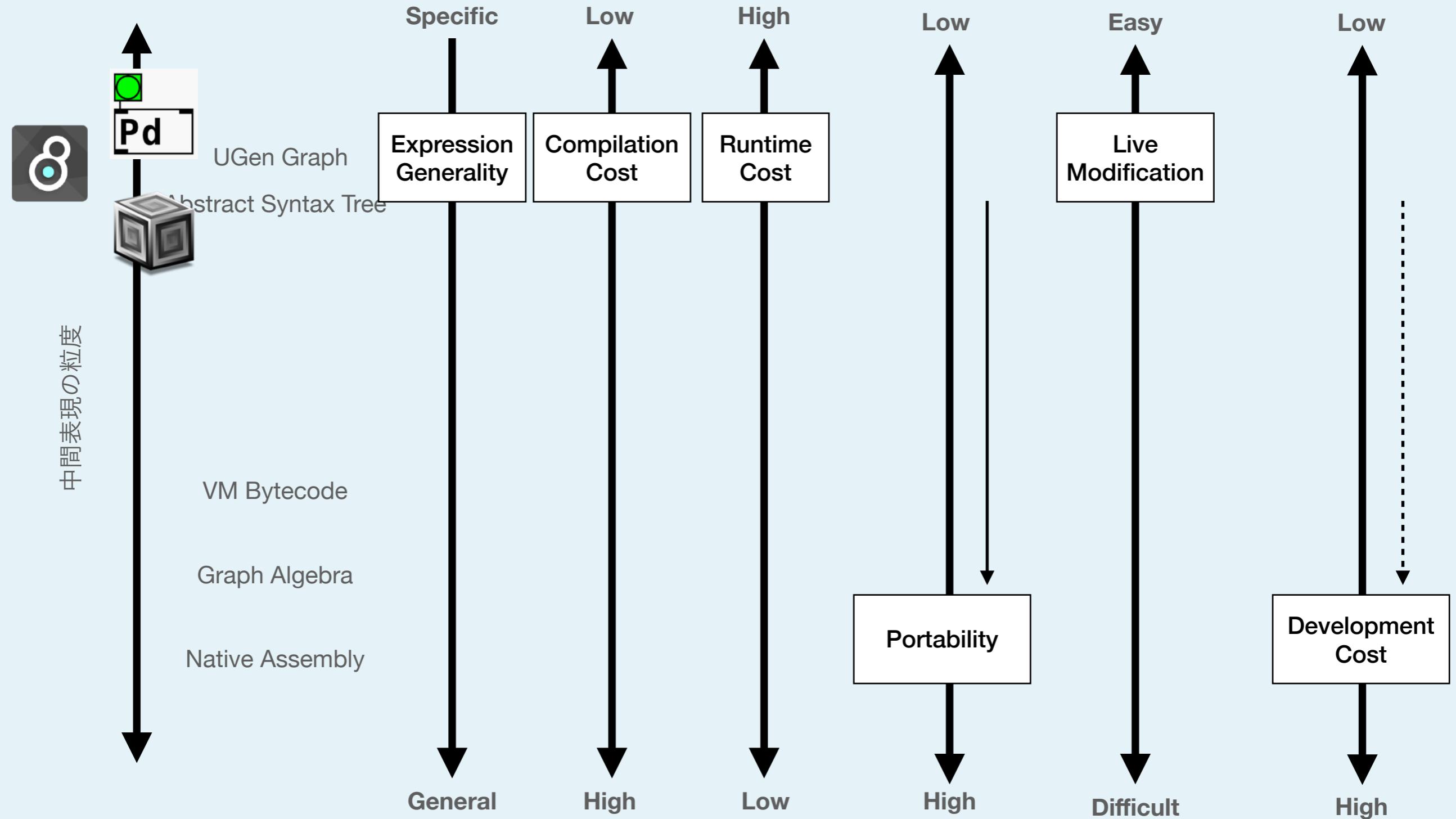
Learnability

Coding Cost

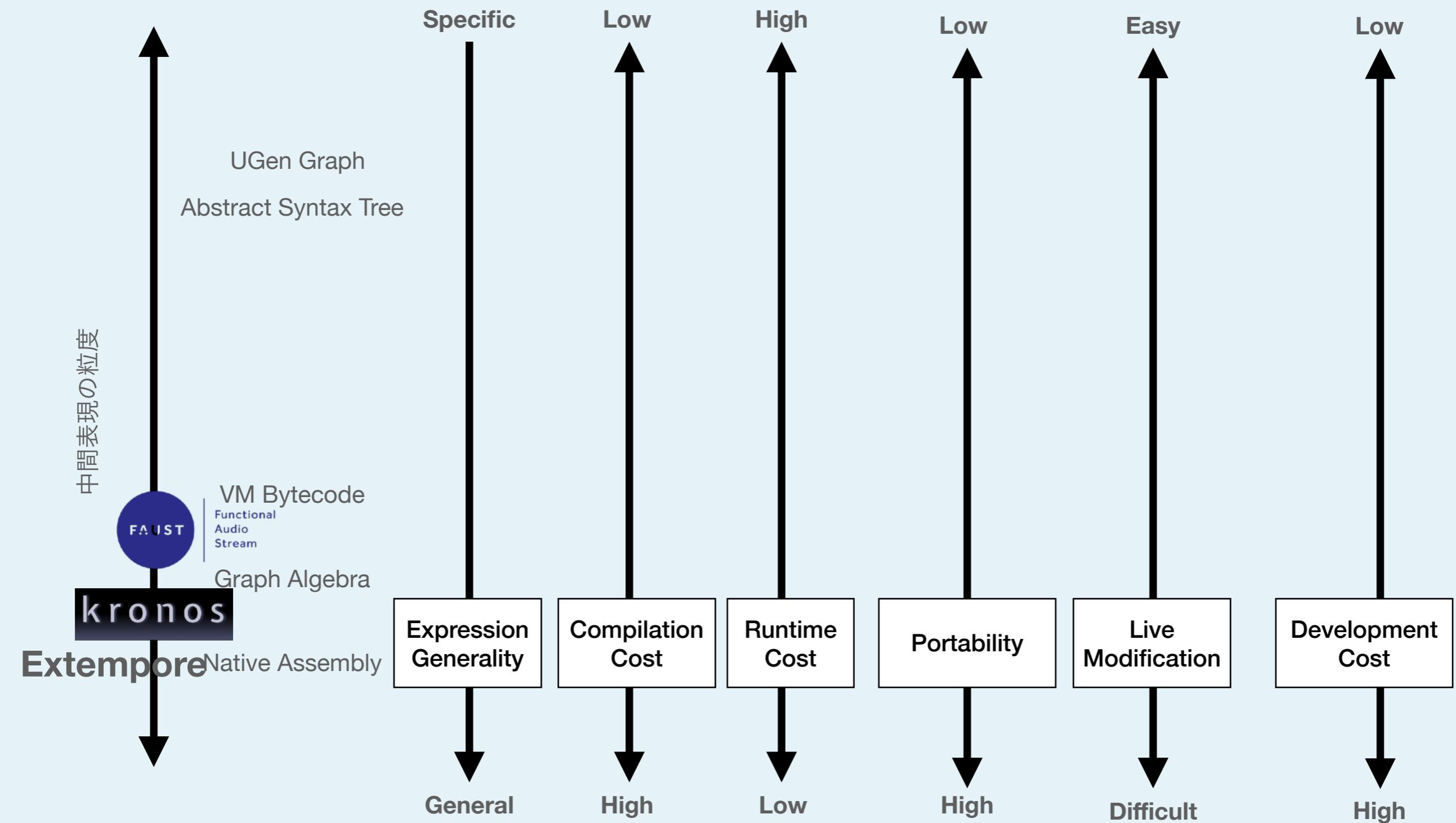
Depends on Syntax design, libraries, IDE, Documentation, Community & etc...



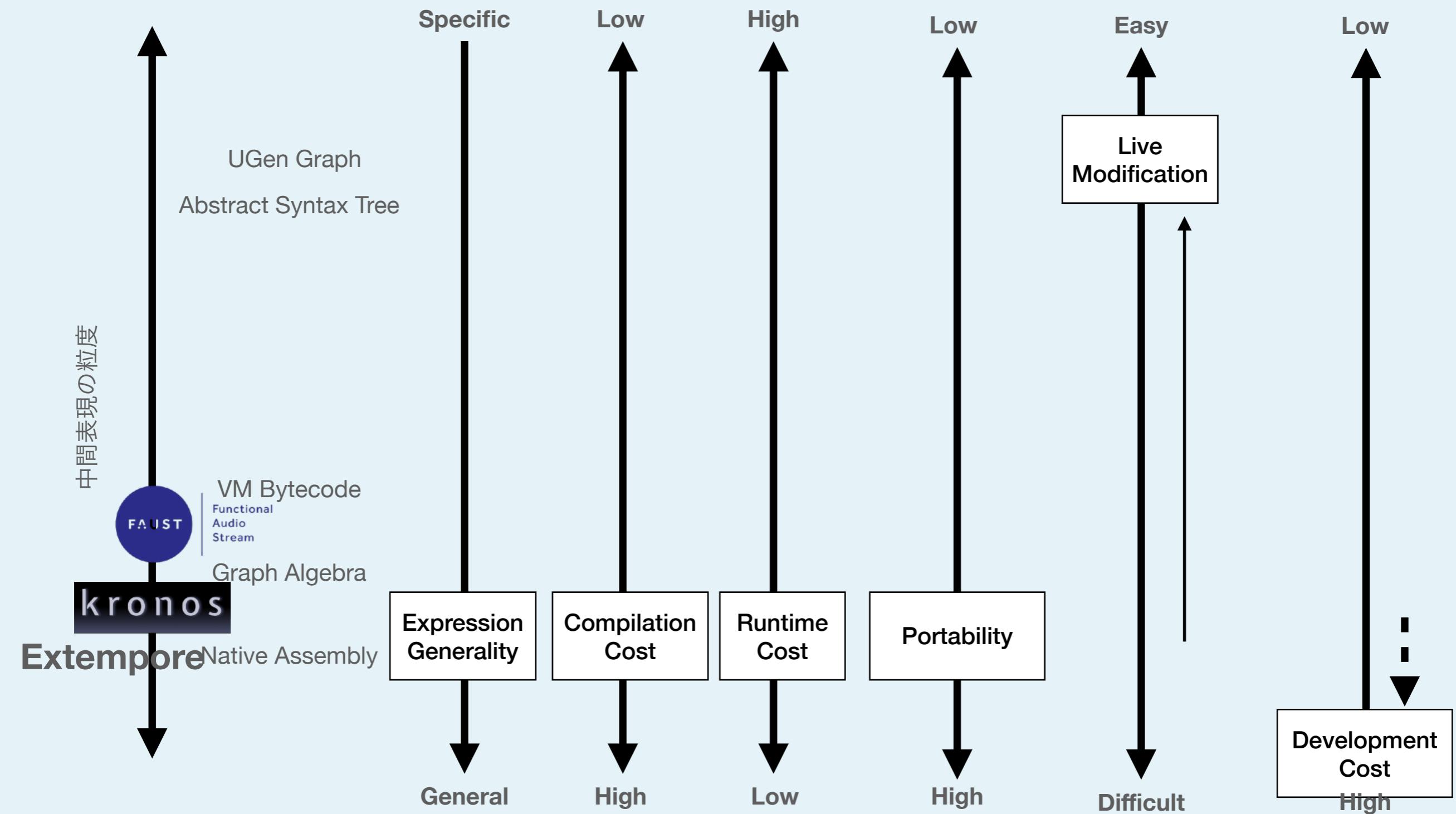
UGenベース：粒度の大きい中間表現はコンパイル時間が不要な代わりに実行コストが大きい
動的変更には比較的強い



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の実装の手間が増える=Development Costは増大



粒度の細かい中間表現を持つ言語は様々なプラットフォームへ移植しやすいが、動的変更が難しくなってくる



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すると、JITコンパイルと呼ばれる複雑な処理を行う必要があるのでDevelopment Costは更に増大

例：Extempore

中間表現の粒度

イベントスケジューリングにはOSの問題もあり完璧な抽象化が存在しない=言語仕様に含めないものも多い

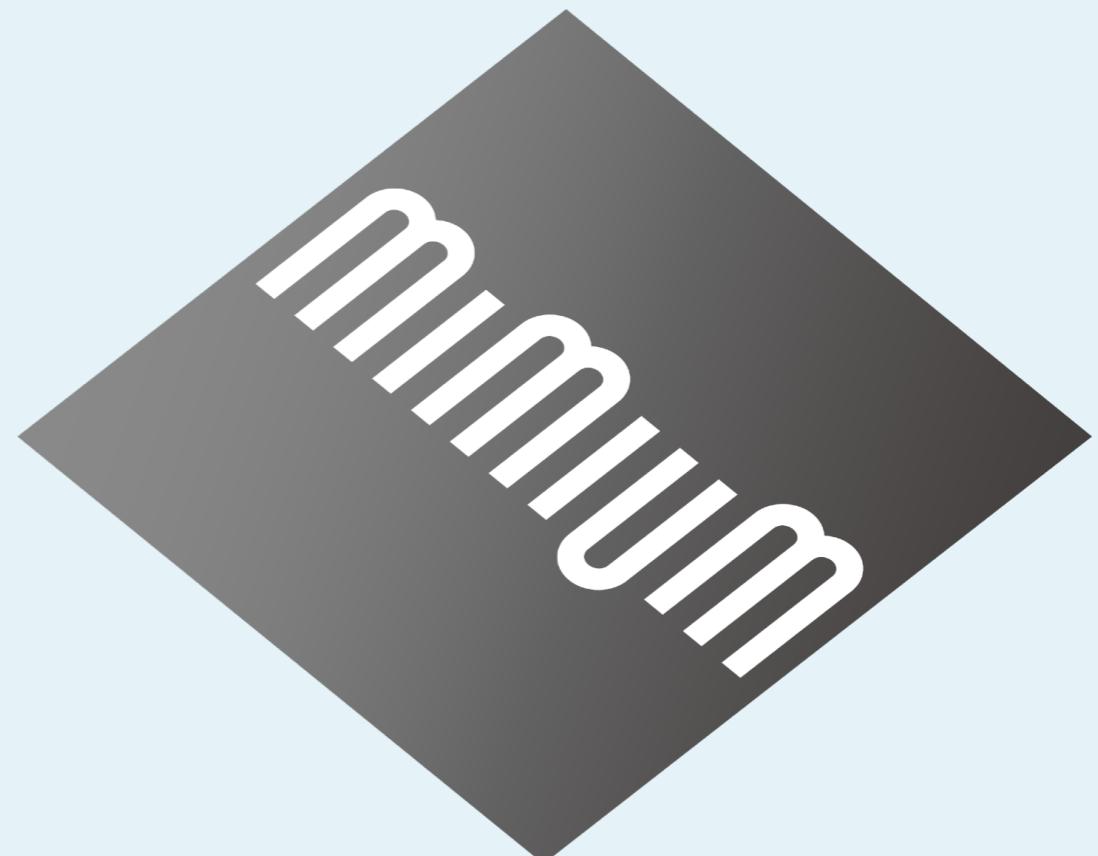


5.PLfMの特徴とmimumに向けた課題

- 2000年代以後の（とくにLive Codingに着目した）言語は、動的変更のしやすさと、UGenという様式から離れた抽象化、もしくはUGenの記述 자체の抽象化を試みたが、それゆえその言語で可能な表現の範囲は局所的である。
- ケイが目指したような、プログラミングを通じた道具の自己拡張性によりフォーカスするならば、UGenよりもプリミティブで、かつ高度な抽象化を行える言語が必要である。
- ただ、根本的トレードオフがあるため、Generalityを優先することは、Live Modificationなどの優先順位を下げる事になる。
- また、Chronicのように抽象モデルだけを考えてリアルタイムで音が出ない、も現実的な問題（スケジューリングなど）から離れすぎている。

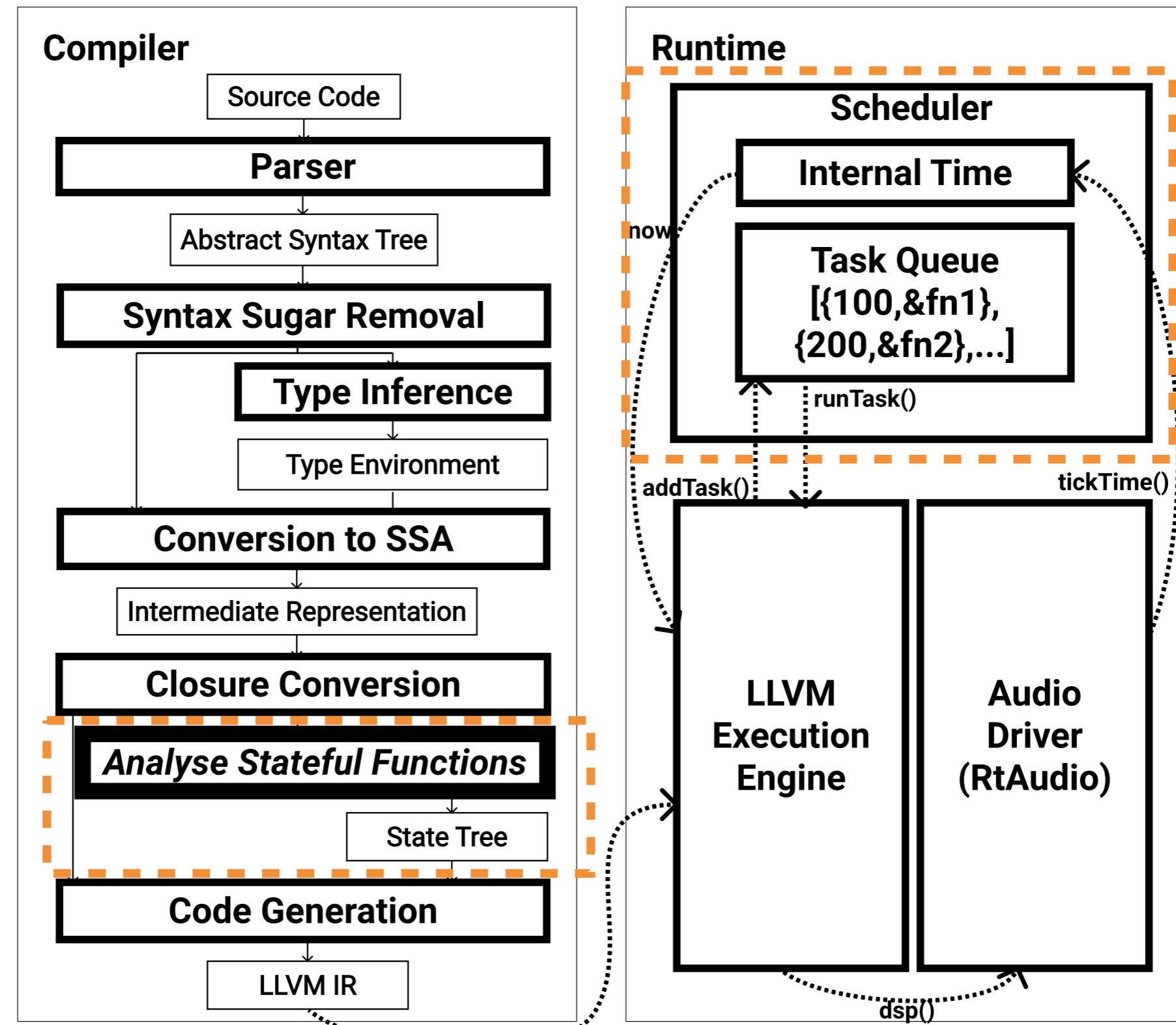
6. mimumの設計

- 関数型の汎用プログラミング言語の設計に、最小限の機能を追加するという方針
 - ハードウェアの隠蔽
 - スケジューリング
 - 論理時間に基づく音声信号の表現



処理系のアーキテクチャ

汎用言語に追加して作られた部分



Design of mimium

既存言語との特徴の比較

	Puredata SuperCollider	Chuck	Extempore	Faust	Vult	Kronos	mimium
スケジューラー	○	○	○			○	○
サンプル精度 スケジューリング		○	○			○	○
UGenの定義		○	○	○	○	○	○
UGenの JITコンパイル			○	○	○	○	○
UGenの関数型 内部表現				Graph	λ 計算	Graph	λ 計算

@演算子によるスケジューリング

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

Extempore

```
fn foo(pitch){
noteOut(pitch,80, 1)
foo(pitch)@(now+48000)
}
foo(pitch)@0
```

mimium

- Extemporeで導入された、**継時再帰 (Temporal Recursion)** という、時間を指定して再帰的に関数を実行するデザインパターンを用いることで、最低限の文法追加で定期的なイベント実行などをあらわすことができる

UGenの代替：内部状態を持つ関数 phasor(0~1を繰り返すノコギリ波)の例

再帰接続の中置演算子 ~

```
phasor(freq) = +(freq/4800) ~ out_tmp
with{
  out_tmp = _ <: select2(>(1),_,0);
};

process = phasor(phasor(10)+1000);
  Faust

fun phasor(freq){
  mem out_tmp;
  out_tmp = out_tmp+freq/48000;
  if(out_tmp>1){
    out_tmp = 0;
  }
  return out_tmp;
}
fun dsp(input){
  return phasor(phasor(10)+1000);
}
```

時間を超えて保持される“mem”変数

Vult

`self`という読み取り専用変数で、

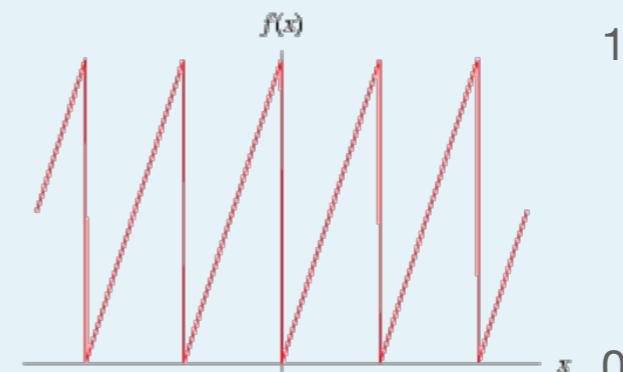
その関数の1単位時刻前の返り値を取得できる

- 実質的にUGenの再帰的接続と等価

```
fn phasor(freq){
  out_tmp = self + freq/48000
  return if (out_tmp > 1) 0
           else out_tmp
}
```

```
fn dsp(input){
  return phasor(phasor(10)+1000)
}
```

mimum



出力波形のイメージ

内部状態付き関数の変換

コンパイル過程における疑似コード

```
fn fbdelay(input:float,time:float,fb:float){  
    return delay(input+self*fb,time)  
}  
fn dsp(){  
    // mix 2 feedback delay with different parameters  
    src = random()*0.1  
    out = fbdelay(src,1000,0.8)+fbdelay(src,2000,0.5)  
    return (out,out)  
}
```

内部状態を引数として渡す形式へと変換する

```
fn fbdelay(state,input,time,fb){  
    self,delay_mem = state           //unpack state variables  
    return delay(delay_mem ,input+self*fb,time)  
}  
fn dsp(state){  
    s_fbdelay0,s_fbdelay1 = state //unpack state variables  
    src = random()*0.1  
    out = fbdelay(s_fbdelay0,src,1000,0.8)  
        +fbdelay(s_fbdelay1,src,2000,0.5)  
    return (out,out)  
}
```



7.議論

- 現状のmimiumの使用における実用上の問題
 1. ベクター処理が不可能 (Faustなど既存の言語と同様の課題)
 2. 命令型の継時再帰と、内部状態付き関数の相性の悪さ
 3. 内部状態付き関数のパラメトリックな複製が不可能
- mimiumのPLfM設計としてのトレードオフの選択 (5章)
- mimiumのPLfM史における位置付け (4章)
- 自由なメタメディアの開発としてのmimium設計 (3章)
- デザインリサーチとしてのmimium設計 (2章)

実用的課題

2: @演算子と内部状態付き関数の組み合わせ

```
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()
```

nという変数を関数自体の実行が終わってもキープする必要がある

- 繰時再帰は基本的に命令型になる
- 信号処理の関数型記述と相性が悪い
- ←のコードのようにできれば組み合わせられそうだが、ローカル変数の寿命管理を考える必要がある

実用的課題

3. 内部状態付き関数のパラメトリックな複製

内部状態をもつfilter関数を複製するような高階関数

```
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  
    }  
}
```

Nがコンパイル時に
わかつていないとself
のためのメモリサイズ
が確定できない

- 内部状態を持つ関数を再帰的に複製したいがNが定数か言語の意味論上では判別できない
- 判別できないと内部状態に必要なメモリのサイズを確定できない
- マクロのように、先にNの部分を評価し再帰関数部を展開してから改めてコンパイルする仕組みが必要

実用的課題

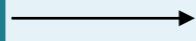
3. 内部状態付き関数のパラメトリックな複製

コンパイル時の処理

ウェーブテーブル生成
UGenグラフ作成

ランタイム上の処理

テーブルを読み出す
グラフを実行

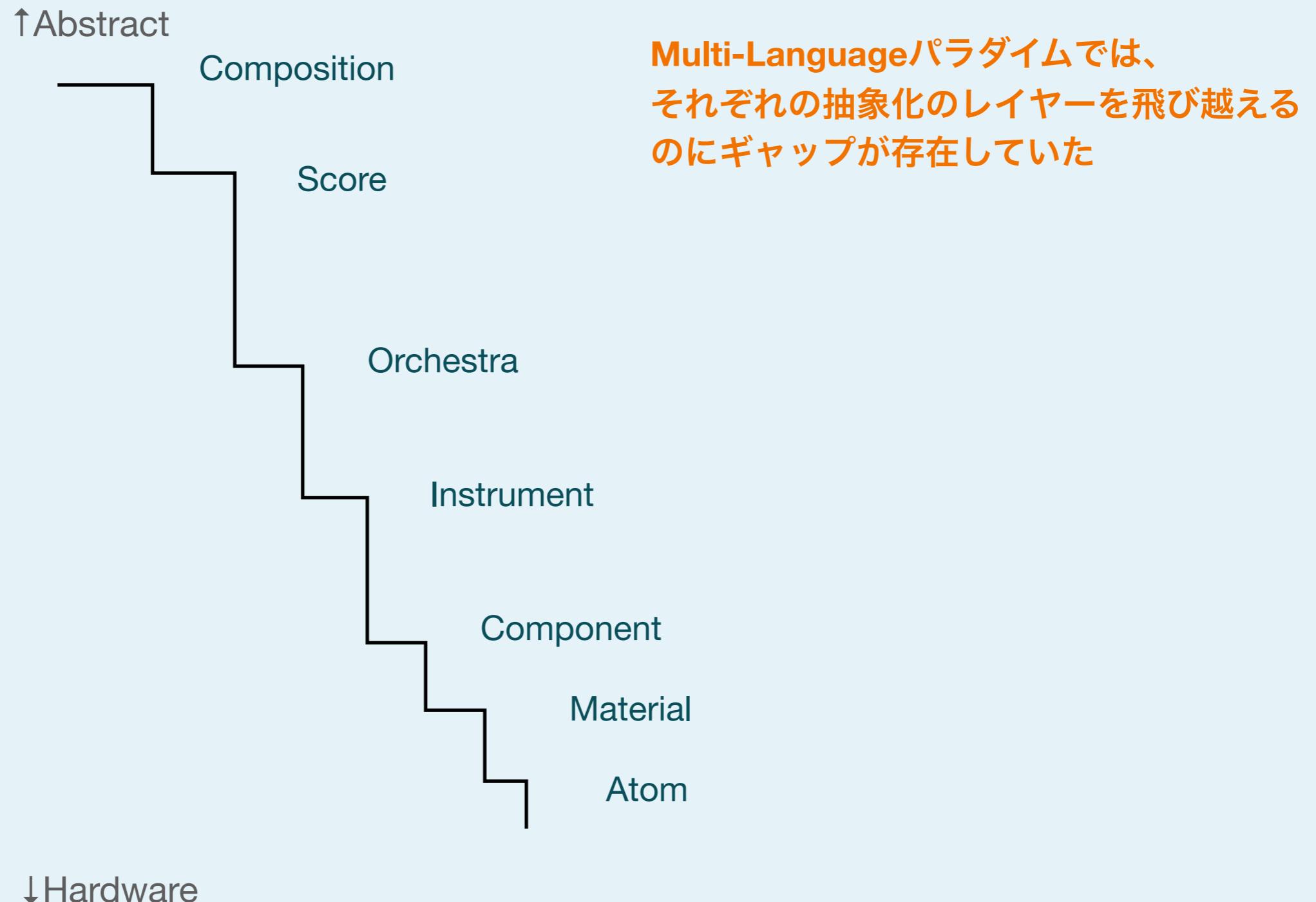


- 音楽のプログラミングはコンパイル時処理とランタイム時処理のような、複数の段階に分けて計算されうるものを一つのソースコードに記述するもの
- MetaML(Taha, 1997)のような、多段階計算と呼ばれる理論が応用できるかもしれない
- 多段階計算はしかも、高機能なマクロのようなものなので、言語上でライブラリとしてDSLを構築するのにも役立てられる(e.g. MacroML(Ganz & et. al 2001))

mimum設計におけるトレードオフの選択

- 当初の予定通り、コードの動的変更には弱い
 - しかし、UGen相当の関数の内部状態が1箇所に集約されるので、ChucKのようなコードの更新時にエフェクトがブツ切れになってしまう問題に対する既存の研究(Reach, 2013)を応用しやすそう
- Generalityと引き換えの開発コストそのものの増大
 - 当初考えていた以上に「とりあえず音が鳴る」状態まで持ち込むのが大変（半年以上）だったのと、モチベーションを保ちづらい
 - ただその多くは開発に用いたC++の難解さにも原因があり、丸ごと書き直すことも視野に
 - Faustなどとコンセプトが被るので積極的に検討してなかったが、MaxやSuperColliderの UGenとして利用できるような広げ方も検討したい
 - インフラは必ず既存のインフラを用いて広がっていく（携帯電話の基地局が給水塔に建てる）というメタファー（Parks and et al. 2015）

プログラミング体験のギャップ



プログラミング体験のギャップ



mimiumでは、各レイヤー間の抽象度のギャップは下げられたかもしれないが、結果として言語そのものの開発過程はほとんど汎用プログラミング言語のそれに近い。

これを開発するモチベーションはどう担保するべきか？

言語処理系自体の開発

mimumのPLfM史上の位置付け

- “音楽のための最低限の抽象化”を現代の状況で改めて再検討
- 例えばTidalCyclesのPatternとしての音楽のような、既存の様式と異なる音楽の抽象化方法を、 mimum上で構築することができるか？
- →データ型の抽象化の性能などの限界で、まだ厳しい。mimum上でDSLを作る = 意味論の自己反映（Reflection）がないと難しい
 - 演算子オーバーロードや、マクロなどの機能追加で可能になるかも
 - 最終的には、音楽のための言語であるけれど、セルフホスト = ある言語の処理系をその言語自身で記述できることになる必要性？
 - PLfMでも、例えばExtemporeでFuture Workとして検討されている

自由なメタメディアとしてのmimium

- ・ 「音楽のための最低限の抽象化」にいまだ確固たる解が存在しない
- ・ 仮に解が見つかったとしても、OSのスケジューラーのような、すでに敷かれてしまったユーザーが関与できない時間制御というアクセス不可能なブラックボックスに対して、とりあえず乗っかる形でしか実装不可
- ・ テキスト以外の編集インターフェースの問題。GUIに限らずとも、コードというデータを表示、編集するアプリケーションと、そのアプリケーション自体のインターフェースの自己拡張性はさらなるFuture Work、かつ、PLfMとは異なる問題系かもしれない
- ・ 少なくとも、これらの問題は「技術の意図的な誤用」では辿り着けなかつた収穫ではあり音楽家の実践の一つとして位置付けられるべきものである

デザインリサーチとしてのmimum設計

- ・ ダン&レイビーのスペキュラティブデザインにて、そこまで注目されていなかった「Design through Science:デザイナーが多少たりとも科学を実践する」
- ・ ダンらの「Design about Science：科学が及ぼす影響をデザインを通じて考察する」との違い：Alternative PresentsやPreferable Futureの「提案」でありつつも、実際にツールとして運用してしまうこと
- ・ 研究領域「音楽土木工学」という概念を実装を通じて創出（後述）
- ・ 一方の反省：ベンチマークや形式的意味論定義など、工学的Evidenceを増やすことも工学的アプローチの人にもより提案への理解を促すために重要：コンセプトが強くとも、工学の皮を被る必要

8. 結論と今後の課題

- 各章のまとめと、全体のFuture workに替えて、**音楽土木工学**という新しい学問の必要性の提案

音楽土木工学

Engineering of a Soil and Wood for Music

Civil Engineering for Music

音楽のための市民工学

まだ見ぬイノベーションよりも、今日使われているもの

Not an unseen innovation but a technology in use

応用よりも、基盤と周縁

Not an application but an infrastructure and marginal area

ブレイクスルーよりも、逐次的修復と改善

Not a breakthrough but an incremental repair and refinement

Music Informatics: How to Compute

Algorithms

Machine Listening

リアルな音声合成

音源分離

空間音響

開発環境

配信プラットフォーム

プログラミング言語

プロトコル | フォーマット

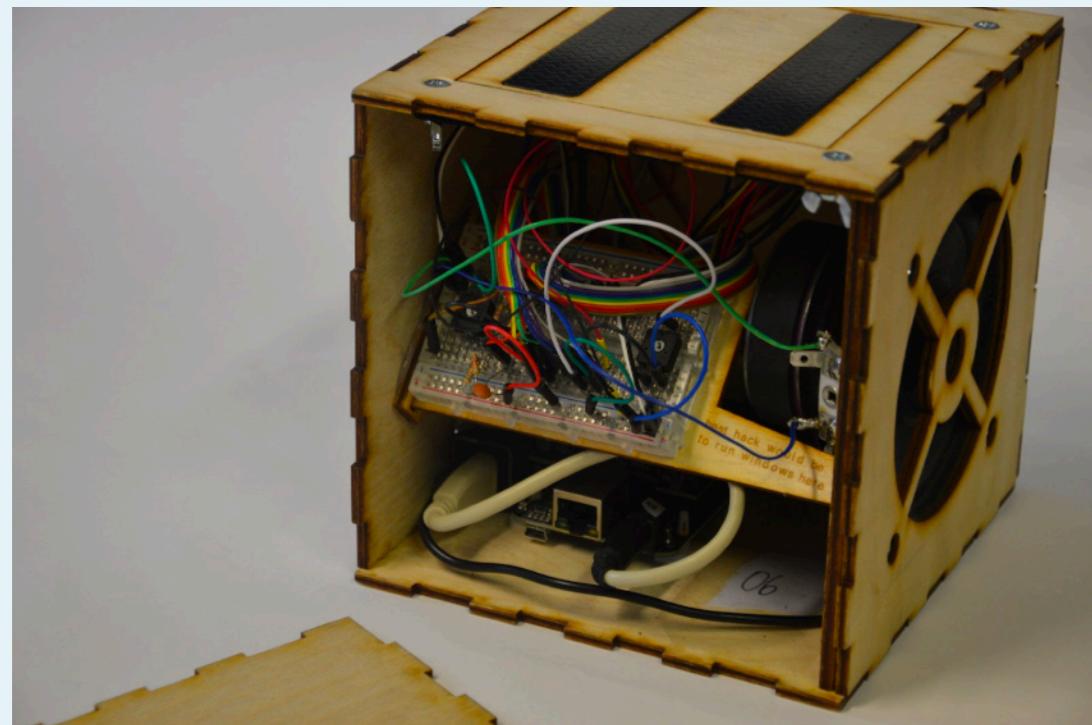
オペレーティングシステム

コンピューター・アーキテクチャ

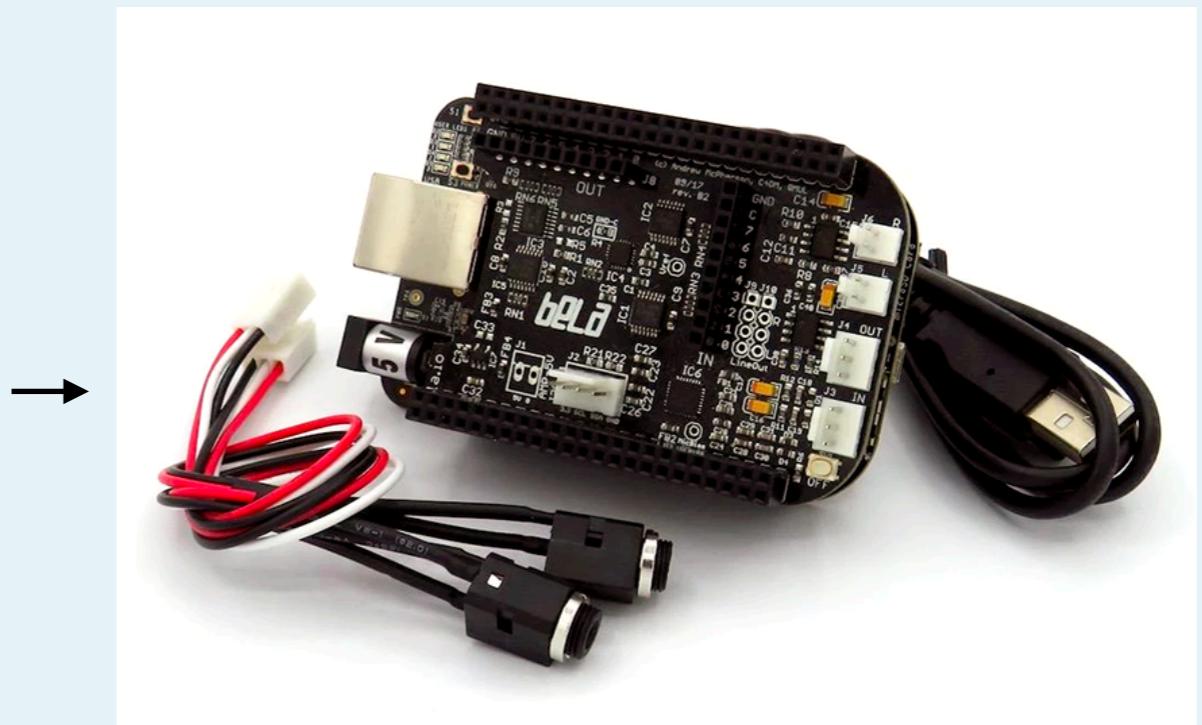
ハードウェア

Civil Engineering of Music Informatics: How to Deploy

ハードウェアとOSの再考



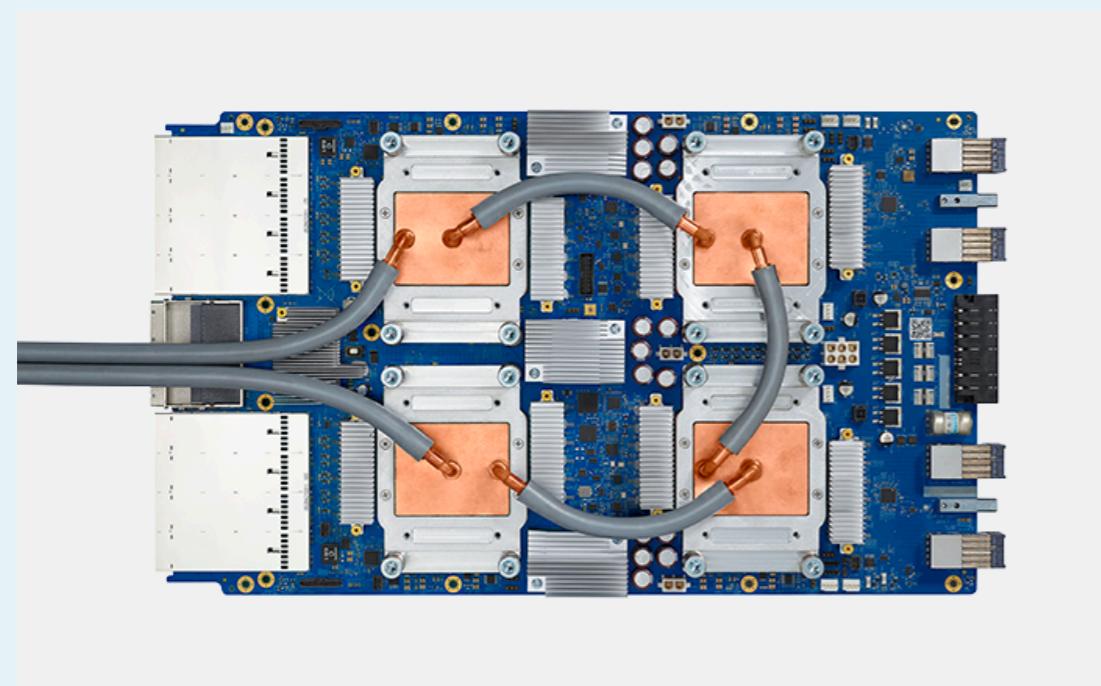
The D-Box by V. Zappi & A. Mcpherson(2015)



Bela, A low-latency audio expansion board for BeagleBone Black(2018)

Xenomaiという、ユーザープログラムがOSのスケジューリングに
関与できる仕組みを採用した、音楽のためのコンピューター

ドメイン特化アーキテクチャ (DSA) の隆盛



TPU、Googleによる機械学習処理に特化したハードウェア(2019)

- 単に汎用的なマルチコアCPUとしての性能向上の限界
- 目的ごとに最適な計算機の構成を設計、製造すればよい(Hennesy&Patterson, 2016など)
- ICチップのパーソナルファブリケーション (Zelooft 2018) といった動きも
- 80年代のチップチューンは（限られたことしかできないとはいえ）ある意味音楽におけるDSA
- Apple H1チップなど、空間音響の処理のための専用チップが導入されているのもこのひとつ→よりプログラマブルにできるのでは？

Music Informatics: How to Compute

Algorithms

Machine Listening

リアルな音声合成

音源分離

空間音響

mimium

開発環境

配信プラットフォーム

Bela

プログラミング言語

プロトコル フォーマット

DSA for
Music?

オペレーティングシステム

コンピューター・アーキテクチャ

ハードウェア

Open Sound
Control

Civil Engineering of Music Informatics: How to Deploy

音楽土木工学という概念の設計

- 工学的なことだけに興味がある人に対しても、あまり着目されていないけど研究しがいのある研究領域を提示する
 - ...という体で工学の皮を被り、ありえるかもしれない現在への想像を取り戻す、政治的行動としてのテクノロジー開発へ徐々に引き込むデザイン・アクティビズム
 - 音楽的なモチベーションだけではmimiumのようなインフラ言語それ自体の開発がますます音楽の話題から離れていく問題に対する一つの実験的解でもある
 - 単なる意図的技術の誤用（失敗の美学）でも、ありうる未来を想像し議論するためのプロトタイプでもなく、実際に横道へと入り込み、けもの道というインフラストラクチャを作る

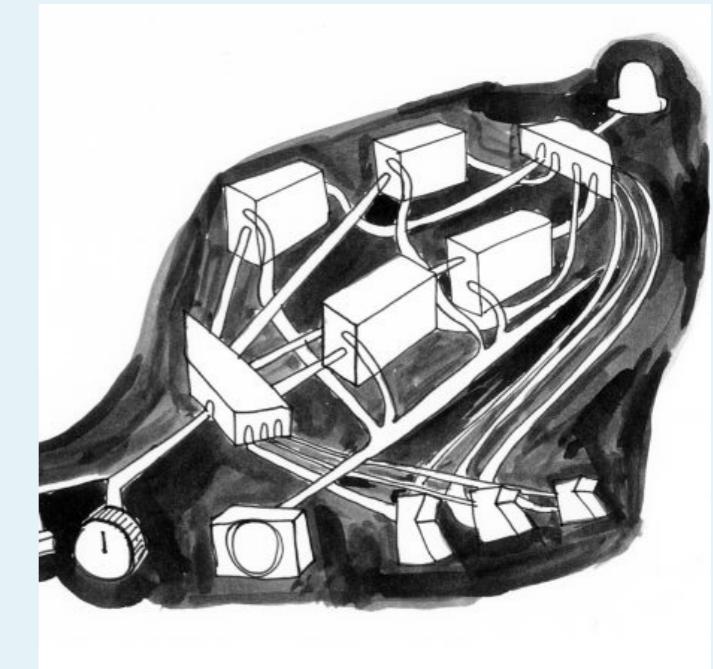


“コンピューターを手で作ることで、オルタナティブな過去、そしてそれが照らし出すオルタナティブな現在とオルタナティブな未来という、テクノロジーとの異なる関係性を想像するインスピレーションを得られた。”

“都市はよく、コンピュータと似て、ブラックボックスの中に封じ込められ、暗号化され、抽象化される。都市の住民は、空間がどう作られ使われるかを限られた範囲でしか制御できないし、私的に保有される公共空間もその意味でまたブラックボックスである。基礎的な部品からわたしたち自身のコンピュータをゼロから作ることで、（自己を）回復するためのオルタナティブな都市空間の創造を想像できはしないだろうか？”



“都市は、コンピュータと似て、美学的熟考のための中立的対象ではない。そうではなく、競合し合う政治性と、危うくて不安定 (precarious) 生活とが置かれた場なのだ。”



Open Circuit, Open City, Taeyoon Choi, 2016

<http://taeyoonchoi.com/2016/12/open-circuit-open-city/> , Accessed 2021/07/13

音楽土木工学

(けもの道を作る)

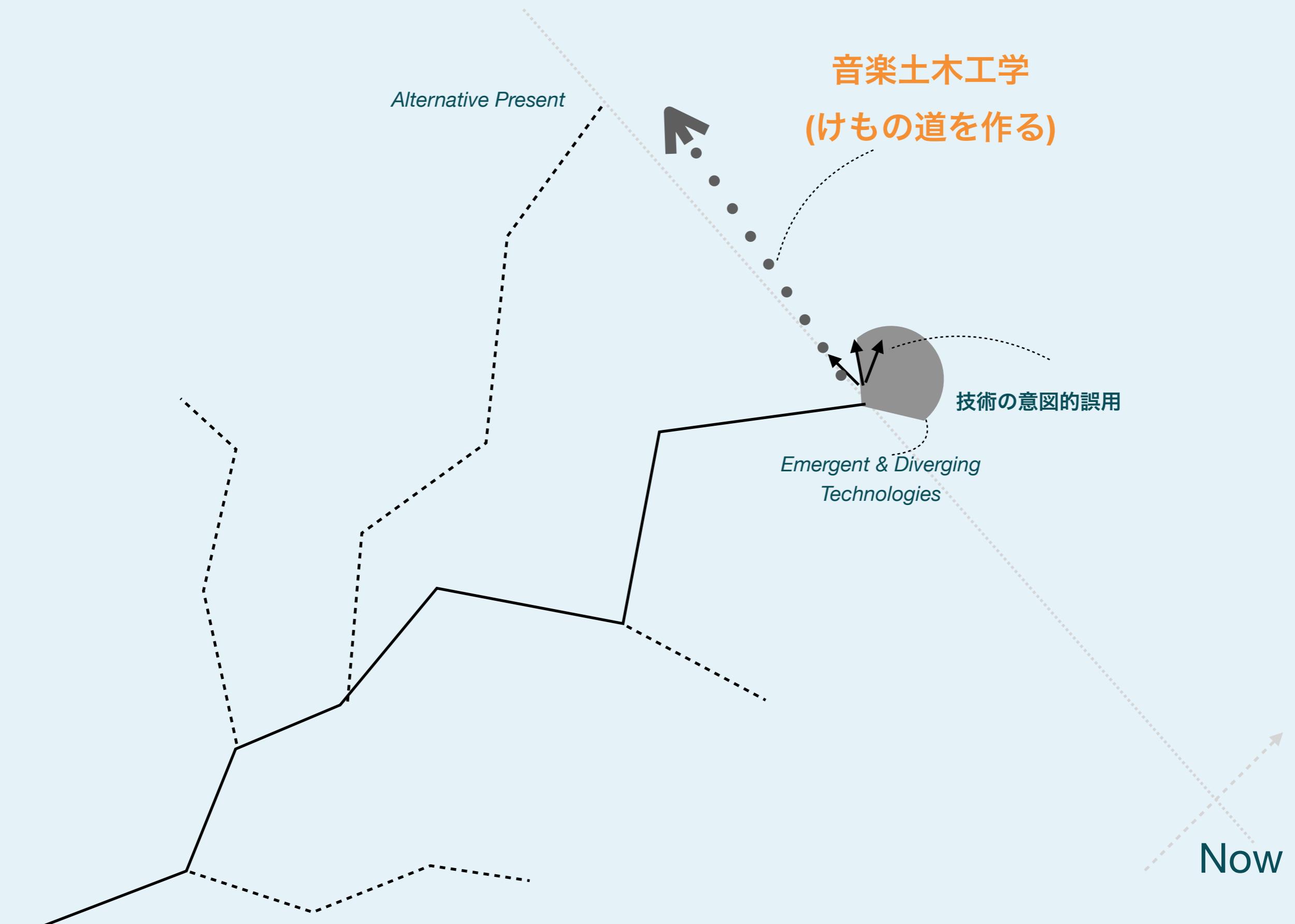
Alternative Present



技術の意図的誤用

*Emergent & Diverging
Technologies*

Now



(Auger, 2010) (Buechly, 2011)を参考に新たに作図