東京大学
情報理工学系研究科 創造情報学専攻
修士論文

無意識を表現に取り入れるアートプログラミングのための統合実行環境「Delevative」の開発

Development of an integrated execution environment for art programming to express unconsciousness

佐藤輝年
Akitoshi Sato

指導教員　千葉 滋 教授

2019 年 1 月

概要

本研究ではプログラムの編集過程に含まれる無意識的動作を活用したアート制作のために，削除文字を含む文字列をソースコードとして実行する統合実行環境を開発した．プログラムの編集過程を活用することで，自動筆記，自動描画のように，アーティスト自身の無意識的動作から，無意識を具現化するアート作品をプログラミングによって制作することができる．開発した統合実行環境は，削除文字を含むソースコードを保存するエディターと，字句定義をカスタマイズできる構文解析器，プリミティブ関数をカスタマイズできる実行器に分かれている．字句定義とプリミティブ関数をカスタマイズすることによってアーティスト自身の無意識的振る舞いの具現化方法を定義することができる．本研究では開発した統合実行環境を用いて，削除文字を含めることで取り出せる無意識的動作を活用したアート作品を制作した．

Abstract

We developed an integrated execution environment that executes character strings including deleted characters as source code for creation of art by unconscious actions included in the editing process of program. By the editing process of the program, it is possible to artistic works embodying unconsciousness from the artist's own unconscious movement like automatic writing and automatic painting by coding. The developed integrated execution environment consists of three windows. One is an editor that saves the source code containing the delete character, the second is a parser that can customize the lexical definition, and finally a window that can customize the primitive function By customizing the lexical definition and the primitive function, the artist can define a method of embodying his own unconscious behavior. And we created an artwork that exploited unconscious behavior that can be retrieved by including deleted characters using the developed integrated execution environment.

目次

第 1 章

はじめに

無意識を表現するアートにはミクロに表現するアートとマクロに表現するアートがある．ミクロに表現するアートは，自動描画のように，無意識の形跡をその瞬間毎に残すアートであり，マクロに表現するアートは，サルバドール・ダリの《記憶の固執》のように，無意識に対する意識的な解釈をまとめたアートである．理想的には，無意識を表現するにはミクロかつマクロであることが理想であると私は考える．なぜならば，ミクロのみの場合無意識の形跡を残しているだけで，表現とは言えず，またマクロのみの場合，実際の無意識の形跡を使用しておらず，表現されているとは言えないからだ．ミクロかつマクロである条件は以下のとおりである．

- ミクロ条件:表現の対象の時間帯の無意識的行動結果を記録する．
- マクロ条件:無意識的行動結果の記録すべてを抽象化した表現をアーティスト自身が意図的に定義する．

本研究ではこのミクロ条件とマクロ条件を満たすアートとして，無意識的プログラミングという新しいアートを提案する．無意識的プログラミングはプログラミングを用いたアートで，ミクロ条件は，無意識的にキーボードをたたき，時間や打鍵圧などの，メタ情報を含む，キーイベントの履歴を保存することで実現し，マクロ条件は，キーイベント配列をソースにした構文解析器をアーティスト自身がカスタマイズすることで，抽象化を行い，構文解析の結果，得られた構文木を実行する実行器をアーティスト自身がカスタマイズすることで表現を定義する．

無意識的プログラミングの実現における課題は，マクロ条件における．構文解析器の実現にある．一般的に取り扱われているキーイベント配列は既存の構文解析手法で解析することができない．なぜならば，既存の構文解析手法は，ソースコードが構造を表現する形に記述された文字列であることを前提としているのに対して，一般的なキーイベント配列は時系列順に並べられており，そのキーイベント列自体が構造を表現していないためである．そこで，本研究で開発した，無意識的プログラミングのための統合実行環境 Delevative では，入力文字に加えて削除文字 (バックスペース) を加えた，削除文字付き文字列をソースコードとして用いることで，この課題を解決した．削除文字付き文字列は入力文字と，削除文字 (バックスペース) を，打鍵時点でのカーソル位置によって並び替えて保存する文字列である．既存の構文解析器

は，ソースコードが，構造的に意味のある形で記述された文字列であることが前提とされているため，時系列ベースで保存された，一般的なキーイベント列を構文解析することは出来ないが，削除文字付き文字列を用いることで，既存の構文解析手法の知見を利用した構文解析と実行を可能にした．

削除文字付き文字列の字句定義にはネスト構造を扱うことができる．PEG を採用した．プログラミング言語における字句定義は正規文法を用いて記述されることが一般的だが，削除文字付き文字列に含まれる削除文字と削除される入力文字との対応関係がネスト構造になっているため，削除文字付き文字列の字句定義をネスト構造を扱うことが出来ない正規文法では定義することができないためである．

Delevative は Editor,HistoryViewer,TokenEditor,PremitiveEditor,ExecutionWindow の 5 つの機能からなる．Editor と HistoryViewer はミクロ条件のための機能である．Editor は無意識的にキーボードを打ちこみ，削除文字付き文字列という形式でキーボードイベント列を保存する機能で，HistoryViewer は，削除付き文字列を表示する機能である．一方で，TokenEditor と PremitiveEditor はマクロ条件のための機能である．TokenEditor は，削除文字付き文字列をソースとする字句定義を編集する機能で，PremitiveEditor は削除付き文字列の情報を利用したプリミティブ関数を定義する機能である．最後に，ExecutionWindow は実行のための機能で，記録された削除文字付き文字列と，アーティストがカスタマイズした，構文解析器，実行器を用いてアプリケーションを実行する．

Delevative を用いた無意識的プログラミングの手順は 5 つからなる．第 1 の手順は無意識的にキーボードを叩くことで，Editor 上で文字列を打ち込むことで行われる．第 2 の手順は無意識的行動結果からプログラム構造を切り出すことで，HistoryViewer で履歴を見ながら，Editor で無意識的に打たれた文字列を削除しながらコードをプログラムの形に書き直すことによって行われる．第 3 の手順は構文解析器をカスタマイズで，HistoryViewer を見つつ，TokenEditor を用いて，削除文字付き文字列として残ったプログラムの字句定義を行うことで行われる．第 4 の手順は実行器のカスタマイズで，PremitiveEditor を用いて，実行時に参照されるプリミティブ関数の定義を行うことで行われる．最後の手順は実行で，削除文字付き文字列として記録されたプログラムを，カスタマイズした構文解析器，実行器を用いて実行する．

この際，Delevative のシステム上では，削除文字付き文字列や，字句定義の情報，プリミティブ関数の情報を保存するサーバー, 字句解析器, 構文解析器, 実行器が動作しておりシステム内では，Editor で打ち込まれたキーイベントを，削除文字付き文字列の形で保存し，字句解析器は，アーティストが定義した字句定義の通りに，履歴をトークンに分割し，構文解析器は，Delevative で予め定義された文法通りに抽象構文木を作り，実行器は，アーティストが定義したプリミティブ関数を参照しながら，実行を行う．字句解析器で削除文字付き文字列の情報をトークンに変換するため，構文解析器は履歴一切見ずに構文解析を行うことができる．これにより，アーティストはプログラミング言語ごとカスタマイズすることなく，字句定義のみのカスタマイズで構文解析器をカスタマイズすることができる．また実行器は実行時に抽象構文木から，プリミティブ関数のみがトークン内の削除文字付き文字列を参照できる形にしている．これにより，アーティストはプリミティブ関数のみのカスタマイズで実行器をカスタマイ

ズすることができる．

以下，2 章では本研究における関連するアート，研究について述べる．既存のアートには無意識を理想的に表現しているアートがないということ，同等のプログラミングを用いたアートがないこと，無意識的プログラミングの実現の上での課題解決のための取り組みがないことを示すために，無意識を表現する既存のアートや，プログラミングを用いたアートを関連するアートとして挙げ，無意識的プログラミングの実現の上での課題に類似した課題を解決した研究や，アート制作のための研究を紹介する．3 章では統合実行環境 Delevative を用いた無意識的プログラミングの具体的な手順について説明する．削除文字付き文字列の形式について説明した後に，先述した Delevative を用いた無意識的プログラミングの手順は 5 つについて詳細に説明している．4 章は，Develvative を実際に動作するシステムとして実装する上で留意した工夫点について述べる．削除文字付き文字列に対応するために実装した構文解析器の実装や，アーティストが用いる字句定義用の DSL，OSC を用いた外部アプリケーションとの通信機能など，無意識的プログラミングを行うための実用的なツールとして利用するために留意した点について述べている．5 章では，実際に Delevative を用いてアートの作例を紹介する．6 章では，本研究のまとめと今後の課題について述べる．

第 2 章

既存のアートと関連研究

この章では，無意識的プログラミングに関連するアートと研究について述べる．無意識的プログラミングに関連するアートとしては，無意識を表現するために行われてきたとされている，シュールレアリスト達の絵画の手法とプログラミングそのものを用いたパフォーマンスアートであるライブコーディングを紹介する．一方で関連研究としてはアート制作に関連する研究，無意識的プログラミング実現のための技術的課題と関連した研究について述べる．

2.1 シュールレアリストの絵画

2.1.1 自動筆記，自動描画

無意識を取り入れるアートとして自動筆記，自動描画がある．自動筆記，自動描画は固定観念をなくした状態で，無意識的に文章，絵画を制作することによって表現をするアートの制作方法である．この技法はダダイズムとシュールレアリスムという 2 つの流派で行われた．ダダイズムはダダイズム発生前までの権威的なアート評価から逸脱し，アートの表現における自由を求めた流派であるため，ダダイズムでの自動筆記，自動描画は，それまでの言葉のあり方や，絵画の表現方法から逸脱する手段の 1 つとして行われた．一方，シュールレアリスムは，人間の無意識を利用したアートを制作することによって，意識から離れた，超現実的な表現を行うことを目的にしているため，シュールレアリスムでの自動筆記，自動描画は，既存の表現からの逸脱よりも，無意識に働きかけた表現を行う手段の 1 つとして行われた．また，関連する絵画技法としては，ダダイズム，シュールレアリスムに影響を受けたとされている，戦後のアメリカで流行したアクションペインティング [1, 2, 3] がある．これは，キャンバス上で絵の具を垂らしたり，飛び散らせたりする絵画技法である．このアクションペインティングには様々な解釈が存在し，アクションペインティングの名を付けた批評家ハロルドローゼンバーグは，キャンバスや筆はアーティストが行動するための手段であるとみなして，あくまで行動そのものを目的としたアートであると解釈した．[1] 一方で，抽象表現主義の文脈から，アクションペインティングによって得られた絵画は行動過程の抽象化の結果であり，絵画そのものにも表現上の意味を持つのだとすると解釈するものもいた．[2] また，批評家のグリーンバーグはあ

くまでアクションペインティングは戦後のアメリカ独自のアートであり，キュビスムなどのアクションペインティング以前の抽象表現主義のアートとつなげるべきではないと主張している．[3] このアクションペインティングは後のパフォーマンスアートに影響を及ぼしている．

ダダイズムでは，あくまで既存の枠組みから逸脱した表現を用いたアートを目的としているため，無意識を表現に用いることを特徴とする無意識的プロググラミングとの直截的な関連性は薄い．ただし，現時点では，無意識的プログラミングそのものが既存の枠組みから逸脱した表現であるため，ダダイズムの流れを汲み取ったアートと解釈することもできる．一方で，シュールレアリスムにおいては，表現方法に無意識的な表現を取り入れることをその流派の特徴としているので，本研究が目指す無意識的プログラミングとの関連は近い．アクションペインティングにおいては，その表現方法の特徴は類似しているものの，無意識的表現であるか否かはアクションペインティング上では重要視されていないため，本質的には無意識的プログラミングとの関連性は薄い．アクションペインティングに影響を受けて発達したパフォーマンスアートの分野 (後述するするライブコーディングなど) には関連性がある．

2.1.2 偏執狂的批判的方法

無意識を表現する上でのマクロ条件を満たす手法の 1 つとして，偏執狂的批判的方法を紹介する．偏執狂的批判的方法はダリが発明した絵画技法で，あるものを別のものとして見たて，視覚化して表現する技法である．例えば，ダリが描いた絵画《記憶の固執》では，時計を，溶けたチーズと見立て，ぐにゃぐにゃと溶けて歪んだ時計を描いている．これは単に時計という固形なものをチーズという柔らかい物の状態を用いて描くという単なる表現のおもしろさ以上に無意識を表現する上でのマクロ条件を満たすという面でも価値がある手法である．ダリが「柔らかいもの」と「硬いもの」に執着していることを自身で述べていることから，ダリは形のさだまらないぐにゃぐにゃとしたものに対する無意識的な執着を柔らかいものを表現し，形の発揮としていて堅固なものに対する知的な執着を硬いものとして表現したのではないかとと解釈されている．[4] 例えば,《大自慰者》では，事故の性的な欲望や去勢への恐怖を有機的なフォルムとともに描いていると解釈されており [5]，また,《ポルト・リガトの聖母, 第 1 習作》では，当時ダリが興味を向けていた科学に対する表現として，原子力と宗教上の神の力とを関連付け，人物や建築を四角く分解し表現していると解釈されている [6].

ダリは自動筆記，自動描画のように意識的表現を排除するのではなく，こうしたダブルイメージを用いた絵画技法によって，意識的に想像力を働きかけ，自身の根底にある無意識的イメージを蘇らせようとした．[7]

2.2 ライブコーディング

ライブコーディング [8] はその場でプログラマのコンピュータのスクリーンを映しながら，音楽を奏でるようなプログラムのコーディングと実行を繰り返していくライブパフォーマンスアートである．ライブコーディングは，AdrianArad と DaveGriffiths によって結成された

Slub というバンドから始まり，その後，JITLib や Fluxus などの様々なライブコーディング用のライブラリや言語の開発盛んにになり，TOPLAP のような，ライブコーディングコミニティが出来るまでに至った．[9] ライブコーディング向けの開発言語やツールは近年でも開発されている．例えば LiveWriting[10] は，ライブコーディング中の入力履歴を記録し，あとで入力の履歴や，実行の様子を再生することができるようなツールである．一般的な楽器のセッションにおいては，MIDI のような，構造を持ったデータとして演奏を記録することが可能であったが，ライブコーディングにはそれに相当するようなものがなかった．LiveWriting によって，再生途中の他者のソースコードをコピーして自身のライブコーディングのソースに貼り付けて演奏することで，非同時的なライブコーディングのコラボレーションを可能にした．また，Estuary[11] は，Haskell ベースのライブコーディング用言語である TydalCycles [12] を元に作られた共同編集環境である．Estuary はサーバーと Web ブラウザー上で動作し，複数人が同時に同じ Estuary のサーバーに Web ブラウザーを通して入ることによって，同じ画面に複数の入力が共存する形で，TydalCycles の複数人セッションを行うことができる．ライブコーディングを複数人でセッションのように行うことや，ネットワークを経由して同一の処理系上で記述することはあったが，インストールなしに，Web ブラウザーのみで，一般的にライブコーディングで用いられている TydalCycles によるパフォーマンスができるという環境は今までになかった．

こうしたライブコーディングは，DJ などと比べて，より詳細に音楽要素を記述できるようにした他，パフォーマンスアートとしてプログラミングを用いたアートの可能性を見出し，プログラムを超えて，アーティストのコーディング過程そのものを鑑賞させるという新しい表現を見出した．一方で，無意識的コーディングではコーディング過程を鑑賞の対象とすることは重要視しておらず，むしろそのコーディング過程の実行結果への反映させることで，無意識的結果を直接用いて表現することを目的としている．

2.3　アート制作に関連する研究

2.3.1　ライブプログラミング

ライブプログラミングは，プログラム編集中に，編集中のソースコードの一部，あるいは関連するコードをリアルタイムに実行し編集者に提示することによって，プログラム理解を促す分野である．ライブコーディングと用語として似ているが，ライブコーディングがアートの 1 分野であるのに対し，こちらは，ソフトウェア工学の文脈におけるリアルタイムに編集の結果が実行されるような開発環境全般を指す．Seymour[13] は，教育向けのライブプログラミング環境である．ソースコード編集中に，ソースコードの実行過程を図示し，提示することによって，被教育者のプログラム理解を促すライブプログラミング環境である．Seymour では，ミクロな可視化と，マクロな可視化で，実行過程を図示しており，ミクロな可視化では，2 項演算の結果をソースコードの隣に表示するなど，各構文がどのような結果になっているのかを瞬時に見ることが出来る．マクロな可視化では，for 文全体を可視化するなど，プログラム全

体の状況を図で理解することができる．また，Shiranui[14] のように，テストケースをソースコード中に記述することで，ソースコード編集中に自動的にテストケースが実行し，テストの結果をエディタ上に表示するようなライブプログミング環境も存在する．アート向けプログラミング言語 p5.js[15] には Auto-refresh 機能が存在し，ソースコードが編集されるたびに，再度実行されるような機能が実装されている．これは既存のアート向け実行環境にライブプログラミングの機能を追加することでライブコーディング向けの環境としてアップデートしたものである．

p5.js の例のように，既存のアート向け実行環境と，ライブプログラミングの関連性は強いが，無意識的プログラミングとライブプログラミングとの関連性は低い．無意識的プログラミングにおいては，無意識的行動結果の記録や，その後のインタプリタのカスタマイズを時間をかけて行うことが想定されており，リアルタイム性は重要視していないからである．

2.3.2 関数型リアクティブプログラミング

関数型リアクティブプログラミング [16, 17] は，イベントが発生した際の処理をコールバック関数を用いて定義するのではなく，変数一つ一つを値ではなくある種のシグナルとしてみなし，変数の定義式内の他の変数が変化するたびに，自身の値も連動して変化するようなプログラミング方式である．例として Javascript のリアクティブプログラミングライブラリの Flapjax[18] を用いたプログラムのソースコードを図 2.1 に示す．このプログラムはストップウォッチのように，リセットボタンを押してから (リセットボタンが押されていない場合はアプリケーション起動時から) の，経過秒を表示するアプリケーションである．1 行目で，nowB でまず，1000 ミリ秒ごとに 1 が追加されるような値が定義され，2 行目で，startTm は nowB に valueNow メソッドを追加し，式が表示された段階での nowB の値を代入している．その後，3 行目で，DOM 上のクリックボタンを押された瞬間の時間 clickTmsB を設定し，4 行目で，経過時間 elapsedB を定義し，5 行目で elapsedB を DOM 要素に適用している．このようにリアクティブプログラミングでは，elapsedB を nowB と clickTmB の単なる四則演算の結果の代入できるように他の値に依存する値の更新をシンプルに取り扱うことができる．

アートの文脈ではライブコーディングで音楽を演奏するための言語 TydalCycles[12] が関数型リアクティブプログラミングに影響を受けたことで知られている．TydalCycles では，1 小節にどのような音を奏でるかを示す関数を定義し，インタプリタ側が，時間に応じてそれらの関数を自動的に呼ぶことによって，シームレスに音楽の構成を変化させることができる．この機能は Flapjax のように関数型リアクティブプログラミングが時間の概念をシンプルに扱えることから発想を受け実装された．無意識的プログラミングでは，こうしたイベントドリブンな操作は現状想定しておらず，あくまで，実行したあとは最初から最後まで設定するので，無意識的プログラミングの目的には合わない．

```
var nowB = timerB(1000);
var startTm = nowB.valueNow();
var clickTmsB = $E("reset", "click").snapshotE(nowB).startsWith(startTm);
var elapsedB = nowB - clickTmsB;
insertValueB(elapsedB, "curTime", "innerHTML");

<body onload="loader()">
    <input id="reset" type="button" value="Reset"/>
    <div id="curTime"> </div>
</body>
```

図 2.1. flapJax を用いたソースコード

2.4　無意識的プログラミングの実現に関連する研究

2.4.1　Metafor

Metafor[19] は，自然言語で書かれた文章からプログラムを作り出して提示し，使用者にその文章の状況を理解させるツールである．元の英文を MonthyLingua[20] を用いて VSOO（動詞，主語，目的語，目的語）形式に変換しこの構造を元に，宣言や配列，if-then 構造のようなプログラミング構造を表すオブジェクトに変換した後に Python のコードに変換する．ここで生成された Python コードは実行するためのものではなく，あくまで文章の状況を理解させ，ブレーンストーミングを促すことを目的としている．この Metafor は記述した内容をプログラミング構造に変換するためのツールであるという点で，構造的な意味を持たないソースコードからプログラム構造を作り出す無意識的プログラミングのマクロ表現の課題解決のアプローチとして用いることができそうだが，Metafor が理解の対象としているのはあくまで意識的に書かれた文章から状況を理解することを目的としているのに対し，無意識的プログラミングでは無意識的に打たれた文字列を含む編集履歴から，無意識を表現することを目的としているという点で異なる．Metafor はあくまで，文法を元に解析するので，アーティストの主観的な解釈を行うためのものではなく，今回の無意識的プログラミングの実現における課題解決をすることはできない．

2.4.2　編集履歴を扱うソフトウェア工学分野の研究

ソフトウェア工学の分野では，プログラムの保守者にプログラムの詳細な変更情報を提示するために，キーイベント単位の細かい粒度でのソースコードの編集履歴を用いる研究

がある．例えば，星野 [21] らによる Historef[22] では，Eclipse が保存している編集記録を OperationRecorder [23] を用いて抽出した後に，編集記録から無駄なものを削除し意味のある編集の単位に分割することによって，git のような版管理システムを用いたプログラムの差分管理ツールにおける，より詳細かつ正確な分割コミットを実現するツールを開発している．また木津らによる研究 (ソースコード編集履歴を用いたプログラム変更の検出) では星野らの研究と同様に，OperationRecorder を用いた編集履歴から生成されるプログラム変更を推定した後に，そのプログラム変更をグループ化することによって，既存の版管理システムよりも比較的理解しやすい形でプログラム変更を保守者に提示するツールを開発している．これらの研究が用いているソースコードの編集操作の履歴と，無意識的プログラミングで用いるキーイベント列は類似しているが，これらの研究の対象は，あくまで分割コミットやプログラム変更の提示といった，プログラム保守の改善を目的としており，キーイベント列を実行に直接反映させることを目的としていないという点で異なる．また，編集履歴のグループ化において，抽象構文木に紐付ける操作を行っていないため，無意識的プログラミングのための言語処理系を実装では，これらの研究成果を活用することはできない．また，Falleri らの研究 [24] では，編集前のソースコードと編集後のソースコードとを比較することよって，どのような編集が行われたかを推定している．具体的には，編集前のソースコード，編集後のソースコードからそれぞれの AST を作ったあとに，編集前の AST と編集後の AST の上で同じと推測されるノードのペアを見つけ，(マッピング) その後，編集後の AST と同じと推測されるノードのペアを用いて，編集スクリプトを計算する．(編集スクリプトの生成)．この研究のように，編集履歴を記録するのではなく，ソースコードから編集履歴を推定する手法も存在するが，正確ではない上に，あくまで，AST の変化の粒度で編集の形跡を見ることが目的なので，一つ一つの操作の粒度で見つけることはできず，無意識的プログラミングのための言語処理系の開発には，これらの研究成果を活用することはできない．

2.4.3 Pressing

Pressing[25] は一つ一つの入力文字における入力時の打鍵圧をメタ情報に含む文字列をソースコードにするインタプリタである．Pressing では Hakoniwa と呼ばれる仮想環境に対して，Ball や Box のような図形を表すクラスのインスタンスを生成の命令や forward,backward といった位置情報の変化や，moveTo のようなスクリーン座標へ移動させるような命令を与えることができるインタプリタで，各命令実行時にその命令の打鍵圧に応じて，図形の大きさを変化させたり移動速度を変化させることができる．Pressing で扱う打鍵圧の情報に加えて様々な情報をメタ情報として含めることができれば，無意識的プログラミングを実現できるように思えるが，あくまで Pressing では入力文字の単位でのみインタプリタが解釈するため，キーイベントレベルの粒度で実行することができないという点と，インタプリタのカスタマイズができないという点で異なる．特にキーイベントレベルの粒度でのインタプリタを構築する上では既存の構文解析手法では難しいため，その構文解析手法を見出さなければ，Pressing の技術を用いて無意識的プログラミングを実現することは出来ない．

第 3 章

Delevative を用いた無意識的プログラミング

この章では，本研究が開発した統合実行環境 Delevative を用いた無意識的プログラミングについて説明する．無意識的プログラミングは，アーティストが無意識的にキーボードを打鍵した記録をキーイベント列として保存し，そのキーイベント列をソースコードとした構文解析器と実行器をアーティスト自身がカスタマイズして実行するアートである．Delevative 上では，無意識的行動結果としてのキーイベント列を削除文字付き文字列という形で保存した後に，記録された無意識的行動結果から，プログラム構造を切り出した後，構文解析器を削除文字付き文字列を字句解析する上での字句定義を記述することでカスタマイズし，実行器を，実行時に参照するプリミティブ関数を定義することによってカスタマイズし，実行する．

本章では，まず削除文字付き文字列について説明した後に Delevative を用いた無意識的プログラミングの手順を説明する．

3.1 削除文字付き文字列

削除文字付き文字列は入力文字と，削除文字 (バックスペース) を，打鍵時点でのカーソル位置によって並び替えて保存する文字列である．また各入力文字，削除文字にはメタ情報として打鍵時のタイムスタンプを持ち，より詳細な粒度でキーイベント列が記録されている．このメタ情報は字句解析時，プリミティブ関数定義時に参照することができる．既存の構文解析器は，ソースコードが，構造的に意味のある形で記述された文字列であることが前提とされているため，時系列ベースで保存された，一般的なキーイベント情報を構文解析することが出来ないが，削除文字付き文字列を用いることで，既存の構文解析手法の知見を利用した構文解析と実行を可能にした．

削除文字付き文字列の具体例を図 3.1 に示す．通常の文字列上で，`print hello` のように表示されるものが，削除文字付き文字列上では，`print` の `i` と `n` の間に `a` を一度押してバックスペースを打つ文字列になり，途中の入力過程が反映されている．

図 3.1. 通常の文字列と削除文字付き文字列との比較 (←はバックスペースを表す)

3.2 無意識的プログラミングの手順

次に Delvevative を用いた無意識的プログラミングの手順を説明する．無意識的プログラミングの手順にはミクロ条件に基づく表現とマクロ条件に基づく表現がある．ミクロ条件に基づく表現は，無意識的にキーボードを叩くことであり，マクロ条件に基づく表現は，この無意識的に叩かれたソースコードをプログラムの形に書き換えた後に，構文解析器，字句解析器のカスタマイズを行うことである．

構文解析器は，字句解析器のみカスタマイズし，字句から抽象構文木を作る機能はカスタマイズしない．アーティストは Delevative 内の TokenEditor ウィンドウで PEG[26] を用いた字句定義を記述する．また，実行器は実行中に参照するプリミティブ関数のカスタマイズを定義することで行う．アーティストは PrimitiveEditor ウィンドウ内で Javascript を用いてプリミティブ関数定義を記述する．

この節では，図 3.2 のように，無意識的に記述された打鍵履歴を引数に取る `feel` 関数の呼び出しを実行すると，感情に対応した音を再生するプログラムを例に，Delevative を用いた無意識的プログラミングの具体的な手順を説明する．

図 3.2. この節で説明するアート例 (←はバックスペースを表す)

3.2.1 無意識的にキーボードを叩く

まず，Delevative 内の Editor を用いて無意識的にキーボードをたたき無意識的行動結果として削除文字付き文字列を記録する．この手順は，無意識的行動結果の記録のための操作なので，最終的なプログラム構造については考えない．feel 関数を 2 つ実行する例の場合も，ここではそのようなプログラムは考えず，無意識的に図 3.3 に示すようなコードを記述する．1 行目は `k` という文字の入力とその削除を繰り返しながら打ち込み，2 行目は `mmnhyu` という文字が打たれた状態である．

図 3.3. 無意識的にキーボードが打たれた場合の Editor 上の文字列 (通常の文字列) と HistoryViewer 上の文字列 (削除文字付き文字列) の例

3.2.2　プログラム構造への書き換え

次に，記録された削除文字付き文字列をアーティスト自身が解釈し，プログラムの形に書き換えを行う．Delevative 上で設定されたプログラミング言語は，プリミティブ関数呼び出しのみを受け付けているので，記録された削除文字付き文字列を，プリミティブ関数自身か，その引数に割り振られるような形で記述する．

feel 関数を 2 つ実行する例の場合，左下の置き換え前の状態から，右上の置き換え前の状態にソースコードを書き換える．ここで初めて feel という文字列がソースコード上に登場する．1 行目の無意識は知人を示す `typhoon` に対する想いであると捉え，`feel typhoon` と書き換える．2 行目の無意識は，恐怖を感じていると捉え，`feel suffering` と書き換える．この際，`typhoon`，`suffering` という `feel` の対象となる引数に記録された削除文字付き文字列が関連づくものだと解釈し，`typhoon,suffering` に記録された削除文字付き文字列が隣接する形で記述する．図 3.4 が示すように，Editor 上には削除された文字列が残らないが，HistoryViewer 上の削除文字列上には削除された文字列が残る．

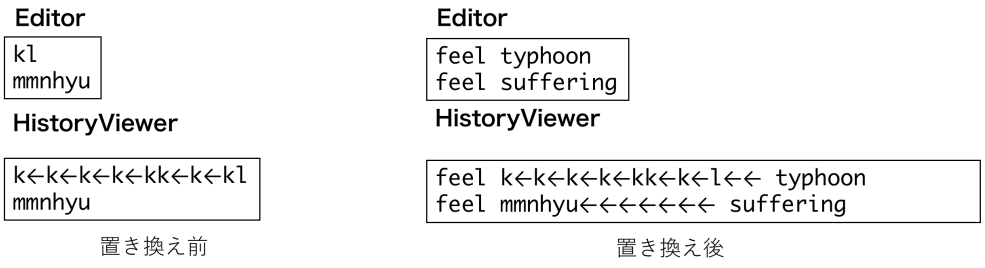

図 3.4. プログラム形へ編集した場合の Editor 上の文字列 (通常の文字列) と HistoryViewer 上の文字列 (削除文字付き文字列) の例

3.2.3 字句解析器のカスタマイズ

次に字句解析器のカスタマイズを行う．削除文字付き文字列と字句との対応関係はアーティストが実現したい表現に依存するため，アーティストが記述する．例えば，feel 関数の例であれば 1 行目の真ん中の k から始まる削除済みの文字列が，feel に着くのか，typhoon につくのか，2 行目の真ん中の m から始まる削除済み文字列が feel に着くのか，suffering につくかは一意に決まらず，表現したいアートに依存するため，アーティスト自身が決定しなければならない．ここでは，feel 関数の引数にとっている，typhoon,suffering という感じる対象自体に削除された文字列との関連があるとし，typhoon，suffering に削除された文字列を結びつけるような字句定義を TokenEditor 内で行っている．(図 3.5)

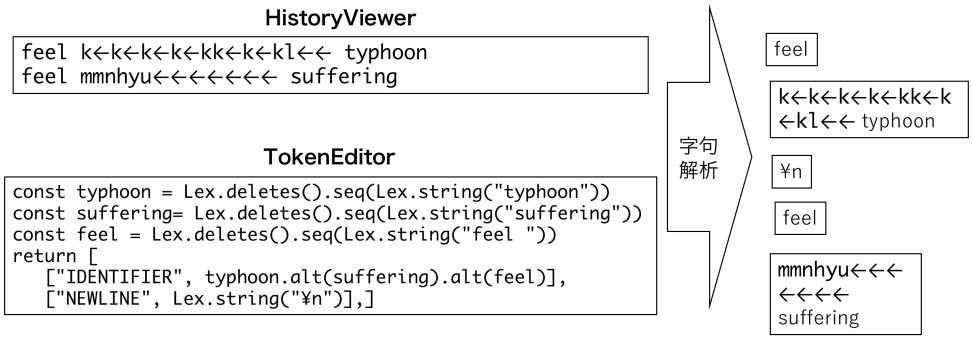

図 3.5. 字句定義例

3.2.4 プリミティブ関数のカスタマイズ

次にプリミティブ関数のカスタマイズについて説明する．構文解析で得られた抽象構文木は削除文字付き文字列が含まれたトークンで構成されているため (図 3.6)，プリミティブ関数の定義では，木に含まれたトークン内の削除文字付き文字列を参照することができる．プリミティブ関数は，構文解析の結果得られた抽象構文木のうち，自身の関数ノード以下に含まれる削除文字付き文字列 history と，自身の関数ノード内の引数ノード arg を引数に取る関数で定義する．feel 関数の例であれば，構文解析の結果左のような抽象構文木が生成されるが，history は関数のノード以下にある削除文字付き文字列，arg はここでは feel typhoon であれば typhoon，feel suffering であれば，suffering に属する削除文字付き文字列がプリミティブ関数の実行時に渡される．(図 3.7)

3.2.5 実行

最後に，記録された削除文字付き文字列をカスタマイズした構文解析器，実行器を用いて実行する．feel 関数の例であれば，この結果として図 3.2 に示すような無意識的に記述された

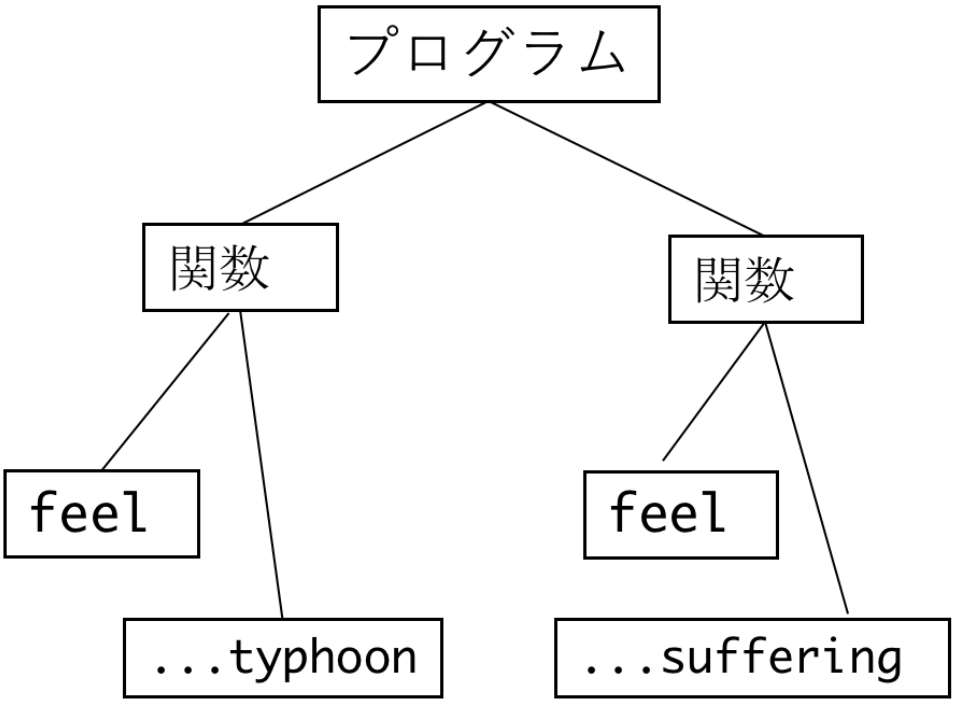

図 3.6. 通常の文字列と削除文字付き文字列との比較

```
[
  functionName:"feel"
  impl:function(history,arg){
    for(var i=0;i<history.length;i++){
      sound(400,history[i].timeStamp.getTime())
    }
    for(var i=0;i<suffering.length;i++){
      sound(arg[i].param,arg[i].timeStamp.getTime())
    }
  }
]
```

図 3.7. 通常の文字列と削除文字付き文字列との比較

打鍵履歴を引数に取る `feel` 関数の呼び出しを実行すると，感情に対応した音を再生するプログラムが実行される．

第 4 章

Delevative の実装

4.1 削除文字付き文字列のための構文解析器とそのカスタマイズ機能の実装

4.1.1 PEG と PackratParsersing

PEG[26] は ParsingExpressionGrammar の略で，CFG に近い文法である．PEG が CFG と異なる部分は OR(|) の代わりに ordered choice(/) を用いることにある．ordered choice は前のパース結果が失敗した場合にのみ後ろの文法を選択する文法であり，これは，ordered choice が OR と違い，前が成功した場合は後ろの文法を選択することはないということを意味している．そのため PEG は CFG と異なり，一つのソースコードに対して解析結果が一意に決定することができる．

PEG をパースするアルゴリズムとして，PackratParsing が知られている．PackratParsing は再帰下降構文解析の一種で, 解析結果を途中でメモ化し, 一度パースした箇所はメモを利用しながら構文解析をしていく手法である．メモ化をすることによって，ordered choice によって生じるバックトラック時に同じ箇所を再パースすることを防ぐことができる．PackratParsing を利用することで，正規文法と同様に O(n) のパース時間で解析することができる．(表 4.1.1)

	ネスト構造	パース時間	メモリ使用量
正規文法	パース負荷	O(n)	O(1)
PEG	パース可能	O(n)	O(n)

表 4.1. PEG と正規文法の表現力と解析に関する比較

4.1.2 PackratParser の実装

PackratParser の実装は，既存のパーサーライブラリを使って生成するのではなくフルスクラッチで制作した．Delevative の実装言語である Javascript のパーサーライブラリーは，文

字列をソースとすることを想定しており，削除文字付き文字列のようのようなオブジェクトの配列をソースとすることを想定していないため，既存のパーサーコンビネーターを利用することができなかったためである．字句定義器は，削除文字付き文字列を指す HistorySymbol の配列を入力にパースし，任意の型の値を返すパーサーコンビネータで実装し，抽象構文木生成のためのパーザーは，字句情報を指す TokenSymbol の配列を入力にパースし，任意の型の値を返すパーサーコンビネータで実装している．(図 4.1) パーザーコンビネータクラス Parser の派生クラスとして，字句定義用パーザーコンビネータライブラリ Lexer と抽象構文木生成用のパーサーコンビネータ Parser とでパーザー 6 種類のパーザークラスを用意した．

```
export class HistorySymbol{
    symbolName:string
    history:History
}

export class TokenSymbol{
    tokenName:string = ""
    tokenSymbol:string = ""
    histories:History[] = []
}
```

図 4.1. HistorySymbol,TokenSymbol

SingleSymbolParser,SymbolTokenParser

字句解析用のパーザーは SingleSymbolParser，抽象構文木生成用のパーザーは SymbolTokenParser である．SingleSymbolParser は，symbolName と合致した HistorySymbol を，SymbolTokenParser は tokenSymbol と合致した TokenSymbol をパースする．symbolName，tokenSymbol はそれぞれ，各削除文字付き文字列，字句の象徴した情報を表す文字列である．symbolName，tokenSymbol を用いることで，文字列情報だけで，入力文字なのか，削除文字なのかを特定できるように実装されている．historySymbol は入力文字なら，入力された文字が，削除文字の場合は backSpace が入る．tokenSymbol はトークン名が入る．

PredicateParser

PredicateParser は特定の条件を元にパースするパーザーである．SingleSymbolParser,TokenSymbolParser では表せない条件を元にパースしたい場合に用いる

SeqParser

SeqParser はコンストラクタ引数に取る 2 つの Parser を順番にパースし，その結果を合成するパーザーである．この際，再帰的に SeqParser を作り続ける many メソッドに備えて，引数に取る 2 つめの値は，パーザーを返す関数を渡し，パース時に遅延評価する形にしている．

AltParser

SeqParser はコンストラクタ引数に取る 2 つの Parser を順番にパースし，前のパーザーが成功した場合は前のパーザーの結果を，後ろのパーザーのみが成功した場合は後ろのパーザーの結果をパースする．パーザーである．

MappedParser

コンストラクタ引数に，パーザーと，パース結果を引数に取って，別の値を返す関数を受け取るパーザーで，コンストラクタ引数で受け取ったパーザーでパースし，成功した場合は，コンストラクタ引数で受け取った関数の引数に入れて実行し，その戻り値をパース結果にするパーザーである．

MemoizedParser

メモ化するパーザーである．コンストラクタ引数にパーザーと，メモのインスタンスを取る．パース時に，メモを参照し，メモ上にパース結果が残っている場合はメモの状態の結果を返す．メモになかった場合は，コンストラクタ引数で取得したパーザーでパースし，その結果を，メモに保存する．メモは，残りの削除文字付き文字列を投げると，パース結果が残る Map クラスのインスタンスで定義する必要がある．

backSpaceSetsParser(**字句定義のみ**)

```
Deletes := ANY(Deletes*)Delete
```

図 4.2. 削除済み文字列の文法定義

字句定義のみで用いるパーザーである．上記の字句定義で定義されている．繰り返し使うことが多いので，クラスではなく，実行時に作成されるパーザーインスタンスとなっており，MemorizedParser を用いて，メモ化されている．

string
任意の文字列をトークンナイズする．文字と文字の間に削除済みの文字列が入るパターンであってもパースする．

これらに加えて各パーサーは以下のメソッドを持つ．

seq
自身と引数にとった，SeqParser を取得する．`this.seq(a)` は `new SeqParser(this, a)` と同様である．

alt
自身と引数にとった，SeqParser を取得する．`this.alt(a)` は `new AltParser(this, a)` と同様である．

many
自身を複数回パースし，その結果を配列にするパーザーである．

maybe
自身をパースし，成功した場合に，引数にとったパーザーでパースを行い，成功したら，自身のパーサーと連結した結果を返す．失敗した場合，自身のパーザーの結果のみを返す．

map
引数にとった関数でパース結果を加工して返すパーザーである．`new MappedParser(this,map)` と同様である．

4.1.3 ユーザー定義 DSL のための Parser ラッパー

独自に生成した PackratParser ライブラリはパース結果の型に制約がない形にしている．これは，字句定義用パーザーで，最終的に字句定義を行うため，また，構文解析用パーザーで，最終的に抽象構文木に変換するため，汎用的であるほうが，都合が良いためである．しかしながら，アーティストが記述する場合には字句定義が書きづらい．例えば，字句定義において，HistorySymbolA の後，HistorySymbolB をパースし，その後 HistorySymbolC をパースするようなパーザーを作り，戻り値の型を HistorySymbol の配列にするには，生の PackratParser ライブラリを記述すると以下の answer のパーザーになる．

```
const AandB:Parser<HistorySymbol[]> = A.seq(() => B);
const AandBandC:Parser<[HistorySymbol,HistorySymbol[]]> = AandB.seq(() => C);
const answer:Parser<HistorySymbol[]> = AandBandC.map(x => x[0].concat(x[1]))
```

こうした煩雑な map 処理を避けるため，アーティストが字句定義するために用いるパーサーは，HistorySymbol の配列を取得し，HistorySymbol の配列を返すもの以外は作らず，トークンへの変換は図 4.11 に示しているような，字句定義とパーザーを関連づけるような配列を返す形にすると良い．図 4.1.3 に，この考えのもと，ユーザーが定義しやすいように作り変えたパーサーコンビネータの実装を示す．

`ParserForUserDefinedToken` が持つ parser は，生の字句定義用 PackratParser ライブラリ上のパーザーインスタンスで，字句定義用の常に `HistorySymbol[]` 型をソースとして，`HistorySymbol[]` 型を返すパーザーになっている．`seq` や `alt` を用いてパーザーを合成する際，合成元のパース結果，合成後のパーザーのパース結果はともにすべて `HistorySymbol[]` 型になるので，変換の際の map 処理は一意に決まることを利用して実装した．実装した合成関数を以下に示す．

seq

自身の定義のパースが終わった後に，引数として受け取ったパーザーオブジェクトの定義をパースして結果を結合するパーサーである．

alt

PEG 上の orderedchoice(/) に相当する文法のパーサーオブジェクトを返すメソッドである．自身の定義のパースが成功した場合は，自身の定義でパースした結果を返し，失敗した場合のみ，引数として受け取ったパーザーオブジェクトを用いてパースし，成功した場合はその結果の配列を返す．どちらも失敗した場合は，パースが失敗する．

maybe

PEG 上の?に相当する文法のパーサーオブジェクトを返すメソッドである．自身の定義のパースが失敗した場合は，パースに失敗する．自身の定義のパースが成功した場合は，引数として受けとったパーザーオブジェクトをパースし，成功した場合は，seq と同様に自身のパース結果と引数のパース結果を連結した結果を返す．失敗した場合は，自身のパースの結果のみを返す．

many

PEG の + に相当する文法のパーサーオブジェクトを返すメソッドである．自身の定義を失敗するまでパースし，一度も成功しなかった場合はパースに失敗する．1 回以上成功した場合は，成功したすべての結果を連結した結果を返す．

各関数の実装内容は，図 4.1.3 に示す．

```
class ParserForUserDefinedToken{
    constructor(parser:Parser<HistorySymbol[]>){
        this.parser = parser
    }
    parser:Parser<HistorySymbol[]>
    seq = (p:ParserForUserDefinedToken) => {
        const newParser = this.parser.seq(() => p.parser)
        .map(x => x[0]
        .concat(x[1]));
        return new ParserForUserDefinedToken(newParser);
    }
    alt = (p:ParserForUserDefinedToken) => {
        const newParser = this.parser.alt(p.parser)
        return new ParserForUserDefinedToken(newParser);
    }
    maybe = (p:ParserForUserDefinedToken) => {
        const newParser = this.parser
            .maybe(() => p.parser)
            .map(x => {
            if(x[1] == null)
                return x[0]
            return x[0].concat(x[1])
        });
        return new ParserForUserDefinedToken(newParser);
    }
    many = () => {
        const newParser = this.parser
            .many().map(
            (x) => {
                return x.reduce((prev,cur) =>
                {return prev.concat(cur)},[])
            }
        )
        return new ParserForUserDefinedToken(newParser);
    }
}
```

図 4.3. アーティストが記述する字句定義 DSL のためのクラス ParserForUserDefineToken の定義

4.2　Editor から HistoryViewer へのカーソル位置の変換

4.2.1　キーイベントからの削除文字付き文字列の更新

Delevative の Editor のキーイベントから削除文字付き文字列を更新する方法を示す．キーボード入力が行われたときに，キーボード入力時の入力されたキー，Editor 上のカーソル開始地点，Editor 上のカーソルの終了地点，現在の削除文字付き文字列，現在の削除文字付き文字列上でのカーソル位置を元に，削除文字付き文字列を更新する．Editor 上のソースコードは，文字削除が起きるたびに，文字数が減っていくが，削除文字付き文字列の場合は文字削除が起きるたびに，履歴数が増えていくため，Editor 上のカーソル位置と削除文字付き文字列上のカーソル位置は別々に管理する必要がある（図 4.4）．削除文字付き文字列上の範囲選択は，削除文字付き文字列を更新する上で不必要なため，削除文字付き文字列上のカーソル位置は開始地点と終了地点を別々に扱わずに，単一の整数の変数として扱う．

まず，Editor 上のカーソル開始地点，Editor 上のカーソル終了地点が同一な状態，つまり範囲選択がなされていない状況からキーボード入力が行われた場合の動作を示す．キーボード入力が，任意の文字列の入力，削除文字 (バックスペース)，範囲選択を含めたカーソル移動であった場合によって処理を場合わけしている．キーボードの入力が任意の文字である場合，現状の削除文字付き文字列上のカーソル位置に図 4.13 の `InputJson` を追加して，削除文字付き文字列を更新する．その後，Editor 上のカーソル開始位置，終了位置，削除文字付き文字列上のカーソル位置をそれぞれ 1 加算する．キーボードの入力がバックスペースである場合，現状の削除文字付き文字列上のカーソル位置に図 4.13 の `BackSpaceJson` を追加して，削除文字付き文字列を更新する．その後，Editor 上のカーソル開始位置，終了位置を 1 減算し，削除文字付き文字列上のカーソル位置は 1 加算する．キーボードの入力が範囲選択を含めたカーソル移動であった場合は，Editor 上のカーソル位置と削除文字付き文字列上のカーソル位置との対応関係を示すハッシュマップを作り，現在の Editor 上のカーソルの終了地点から，削除文字付き文字列上のカーソル位置を再計算する．範囲選択がなされている場合は，範囲選択分だけバックスペースによる削除が行われた後に，入力が行われたものとみなして記録される．

Editor 上のカーソル位置と削除文字付き文字列上のカーソル位置との対応関係を示すハッシュマップの計算方法を図 4.2.1 に示す．まず，各編集履歴に削除済みかどうかのフラグを付けた，`HistoryWithDeleted` の配列を作る．次に，削除済みのフラグは削除された入力文字と，削除文字自身にフラグに true をつける．`HistoryWithDeleted` は，`HistoryJson` の配列を逆から読みつつ，削除履歴が含まれるたびに 1 カウントし，入力履歴にたどり着いた段階で，カウントが 1 以上なら deleted フラグをつけてカウントを 1 引き，カウントが 0 すると，deleted フラグはつけないようにする．今回の実装では削除文字付き文字列にデリートキーが含まれないが，デリートキーが含まれる場合も，同じ操作を正の順番で読み込みながらフラグをつければ良い．

次に，`HistoryWithDeleted` を図 4.2.1 に示すような削除文字付き文字列上でのカーソル位

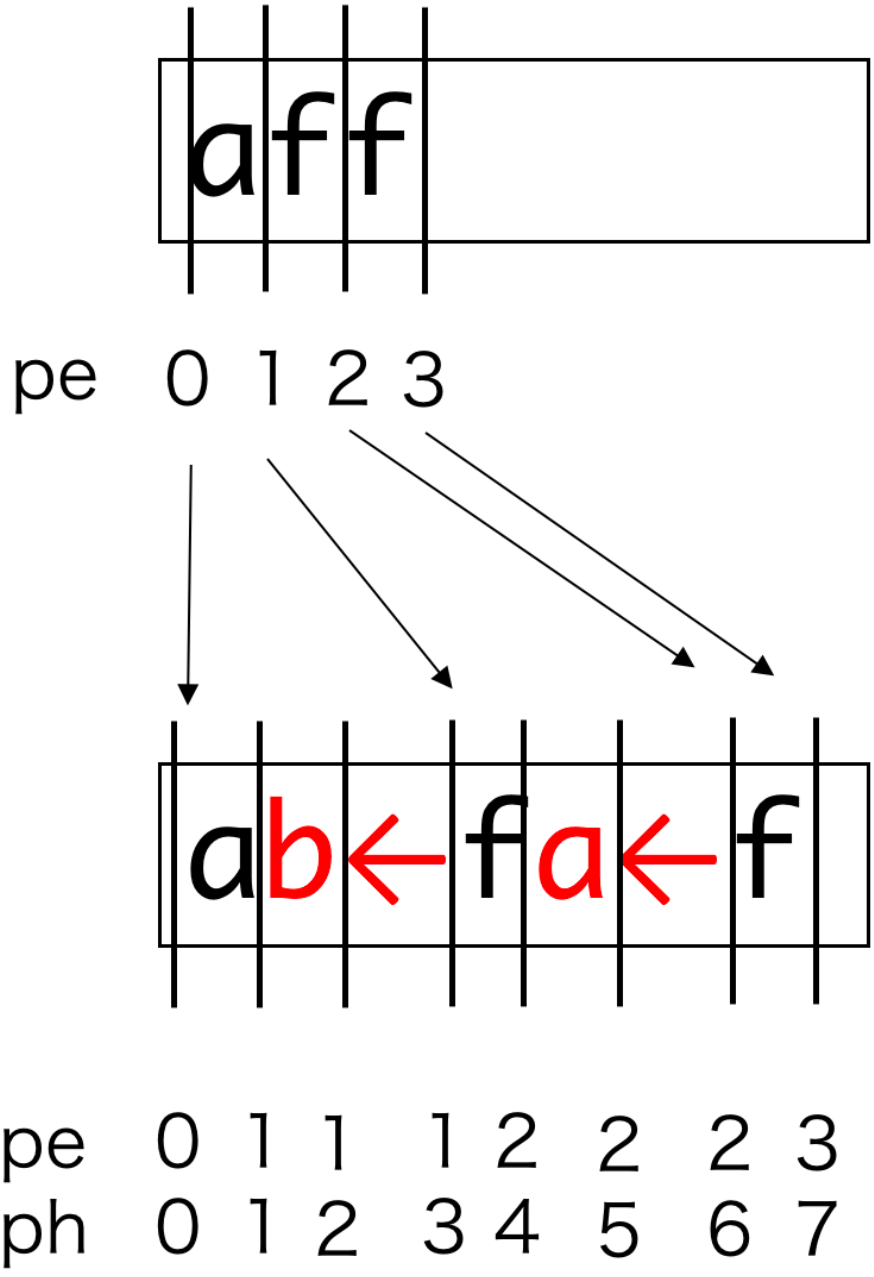

図 4.4. 同じソースでの，Editor 上でのカーソル位置 (pe) と削除文字付き文字列上のカーソル位置 (ph)．(赤文字は削除済みの入力文字と削除文字)．削除文字付き文字列と Editor の削除文字付き文字列とでは，カーソル位置を別で考慮する必要がある．

```
{
    history:{
       {
          kind:” backSpace”
          timeStamp:” 2019-01-06T08:17:32.107Z"
       }
    },
    deleted:true
}
```

図 4.5. HistoryWithDeleted の内部オブジェクト例

置を占める配列にまとめる．削除文字付き文字列上のカーソル位置はカーソルの左右にある文字のペアで構成されており，カーソル位置から見た左，あるいは右に履歴が存在しない場合は empty が含まれる形になる．

```
[
  [empty,a],
  [a, ←],
  [b,c],
  [c,d],
  [d, ←],
  [←,empty]
]
```

図 4.6. 削除文字付き文字列上のカーソル位置を表す配列例（削除文字付き文字列が a ← bcd ← s である場合）

最後に，この削除文字付き文字列上のカーソル位置を表すペアの配列を最初から最後まで走査しながら，ハッシュマップを生成する．ペアの配列を最初から最後まで走査するなかで，カーソル位置から見て左側に削除済みの文字列がある場合のみ，エディター上の位置を加算していくことで計算する．

```
function(historyCurserPosArray){
  const map = new Map<number,number>();
  let count = 0;
  editorCuserPos

  historyCurserPosArray.forEach((curHistoryCurserPos,index) => {
      map.set(count,index);
      const insert = inserts[index][1]
      if(inserts[index][1] != "empty" && !insert.deleted){
          count ++ ;
      }
  })
  return map;
}
```

図 4.7. Editor 上のカーソル位置から，削除文字付き文字列上のカーソル位置を導くハッシュマップの計算

4.3 Electron による実装と状態管理

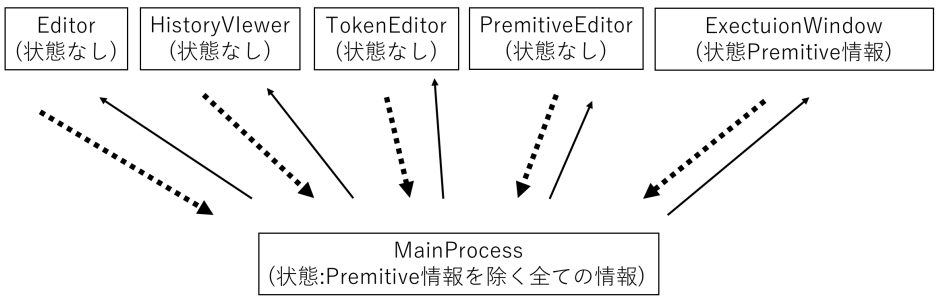

図 4.8. Electron 上での Delevative の状態遷移

Delevative は Electron[27] を用いて実装されている．Electron はデスクトップアプリケーションにおける各ウインドウとして Chromium[28] ブラウザを生成し，ブラウザのレンダープロセスと Node.js で起動しているメインプロセスとの間で IPC 通信を行うことによってデ

スクトップアプリケーションを実装することができるフレームワークである．ブラウザーのレンダープロセスは Chromium 上の HTML+Javascirpt, メインプロセスは Node.js で動作している．

Delevative を用いた Electron メインプロセスとブラウザーレンダープロセスの設計概要を図 4.8 に示す．読み込みや保存のために，メインプロセスに一元的に状態を管理し，構文解析などの演算処理は，メインプロセスが行い，サーバーでは各ウィンドウが持つ情報を一元的に管理している．基本的に各ウインドウは，ウインドウが発するイベントをサーバーに送ることと，サーバーから受け取った情報を元に画面にレンダリングすることのみ行う．ただし，ExecutionWindow のみは，実行結果をウインドウに反映するために，実行する Primitive 関数情報を記憶し，ブラウザ上で実行するように実装している．

4.4　OSC による外部アプリケーション連携の実装

プリミティブ関数の定義に用いる Library として，OSC プロトコルで情報を送信するための sendOSC 関数を提供している．OSC[29] は他のデバイスやアプリケーションと通信するためのプロトコルで，コンピュータを用いたアートで，一般的に用いられている．OSC はトランスポート層に依存しない規格であり，USB シリアル通信などでも OSC を用いた通信が行えるが，一般的にコンピュータアートを作るアーティストは一般的に TCP/IP または，UDP を用いて情報をループバック送信することで，他のアプリケーションと連携した作品を制作しているため，Delevative 内の OSC ライブラリは node-osc[30] を利用し，WebSocket 上で OSC プロトコル通信を実現している．OSC プロトコルは情報をアドレスを指定して送ることである．アドレスは,”/voice” “/sound” “/movie” のような URL パスに類似したアドレス名で指定する．OSC を用いることで，OSC で通信での通信機能が実装されている言語（vvvv[31]・MaxMSP[32]・PureData[33]・Processing[34]・openframeworks[35]）などの機能を利用し，Delevative で制作するアートの表現の幅を広げることができる．

4.5　システム構成

Delevative のシステム構成について説明する．Delevative は Editor，HistoryViewer，TokenEditor，PrimitiveEditor，ExecutionWindow の 5 つのウィンドウからなる．以下で順を追って説明する．

4.5.1　Editor

Editor は TextArea の情報を元に，削除文字付き文字列を保存し，削除文字付き文字列を元に Editor 上の入力文字列とカーソル位置を計算して描画している．キー入力とバックスペース，マウスやキーボードによる範囲選択を受け付けるが，デリートキーは受け付けない．

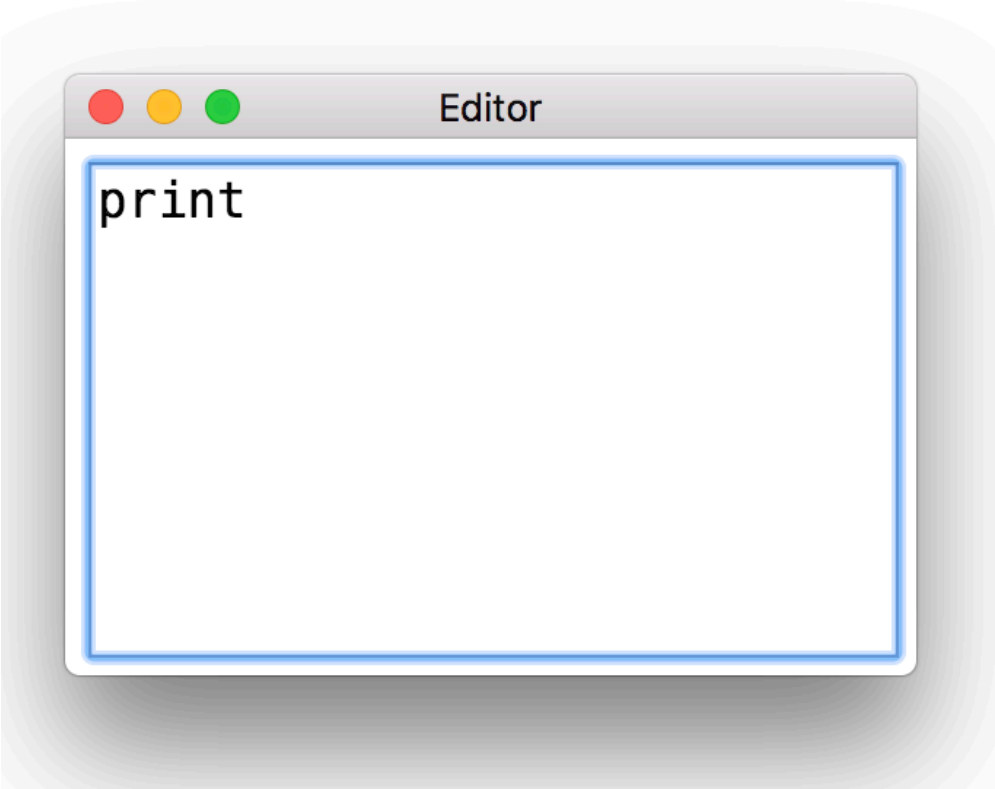

図 4.9. Editor

4.5.2 HistoryViewer

HisotryViewer は削除文字付き文字列を表示する．削除文字付き文字列は入力の場合は，その文字を表示し，削除の場合は←として表示する．HistoryViewer 上では入力をすることはできず，Editor から削除文字付き文字列が更新されるたびに，HistoryViewer の情報も更新される．

4.5.3 TokenEditor

TokenEditor では削除文字付き文字列を字句解析する際の字句定義を記述する．CheckTokenSource で現状の字句定義に問題がないかをチェックすることができる．今回 4.1.2 に示した文法の言語を扱うため，`IDENTIFIER` と `NEWLINE` の 2 つの字句を定義する．TokenEditor 内では，PEG[26] を記述する DSL を用いてパーザーを作り，そのパーザーと字句との対応関係を示すオブジェクトを返す関数を定義することによって字句定義を行う．PEG を記述する DSL は Javascript で記述することができる内部 DSL ライブラリとして用意されている．PEG ライブラリは TokenEditor 上からのみ用いることができる Lib オブジェクト内に含まれた PackratParser インスタンスで記述できる（4.11）アーティストは，トークンごとのパー

図 4.10. HistoryViewer

ザーを PEG ライブラリで定義後に，各パーザーが IDENTIFIER，NEWLINE のいずれかと，トークン定義を持つ配列の配列を返すことで IDENTIFIER と NEWLINE の字句定義を記述する．

Lib に含まれる PackratParser インスタンスは以下のとおりである．各 PackratParser は全て，プログラム全体の削除文字付き文字列を表す HistorySymbol の配列をソースに，パース結果の HistorySymbol の配列を返す．

Lib.deletes

既に削除された文字列をパースすることができるパーザーである．deletes の文法定義については？で示した．また，deletes はバックトラックを多用するパーザーであるため，メモ化がなされている．

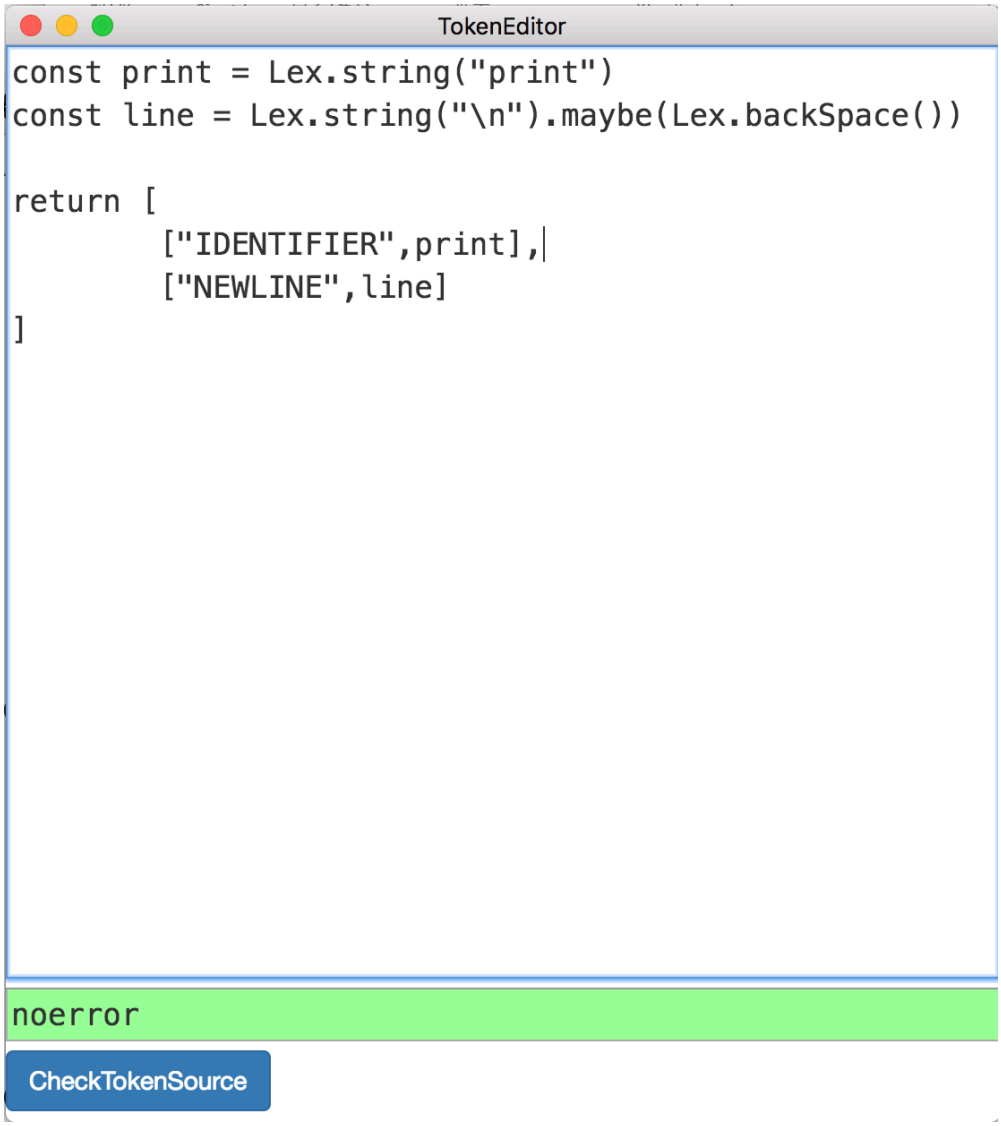

図 4.11. Delivtative の TokenEditor による字句定義

Lib.string

引数に指定された文字列をパースするパーザーである．文字と文字の間に，Lib.deletes が含まれたケースも考慮した文法のパーザーであるので，`Lib.string("hello")` は，`"hello"` という編集履歴と，`"helaa` ←← `lo"` という編集履歴，どちらもパースすることができる．

Lib.backSpace

バックスペースを考慮するパーザーである．

Lib.if

引数に指定された条件を満たした編集履歴をパースするパーザーである．今回作成した Delevative 上では timestamp 情報を用いているため，timestamp を用いたパースを実現することができる．条件は，引数に編集履歴 1 つを受け取り，boolean を返す関数で定義する．

4.5.4　PrimitiveEditor

PremitiveEditor

```
return [
        {
                "functionName":"print",
                "impl":(history) =>
                {
                        var a = document.getElementById("execution")
                        history.forEach((x,i) => {
                                setTimeout(() => {
                                        if(x.kind == "input"){
                                                a.innerHTML = "hey"
                                                Lib.sendOSC({
                                                        ipAddress:"127.0.0.1",
                                                        oscAddress:"/hello",
                                                        portNumber:8000,
                                                        value:300
                                                })
                                        }else{
                                                a.innerHTML = "wow"
                                                Lib.sendOSC({
                                                        ipAddress:"127.0.0.1",
                                                        oscAddress:"/hello",
                                                        portNumber:8000,
                                                        value:200
                                                })
                                        }
                                },i*100)
                        })
                }
        }
]
```

図 4.12. Delivtative の PrimitiveEditor によるプリミティブ定義

図 4.12 は Delevative 上の PrimitiveEditor を用いたプリミティブ関数定義である．プリミティブ関数定義は，プリミティブ関数名を表す `functionName` とプリミティブ関数の実装を表す `impl` を含めたオブジェクトの配列として記述する．プリミティブ関数定義はプリミティブ関数呼び出し全体の部分での `HistoryJson`(入力履歴) の配列である history とプリミティブ関数の引数を表す arg を引数に取る javascript 関数で定義することができる．arg は `ParamWithHistoriesJson`(引数を表すオブジェクト) の配列で，ParamWithHistoriesJson は, 引数の値自体を表す param と引数の入力履歴のを表す HistoryJson の配列である histories によって表されている．HistoryJson の型情報は図 4.13 に示す．入力あるいは削除であることを表す kind と入力された時刻を表す timeStamp が含まれており，入力履歴である場合は，入力された文字列を表す char が代入されている．

```
//入力履歴
{
    kind:” input”
    char:” a"
    timeStamp:” 2019-01-06T08:17:31.917Z"
}
//削除履歴
{
    kind:” backSpace”
    timeStamp:” 2019-01-06T08:17:32.107Z"
}
```

図 4.13. HistoryJson の定義

4.5.5 ExecutionWindow

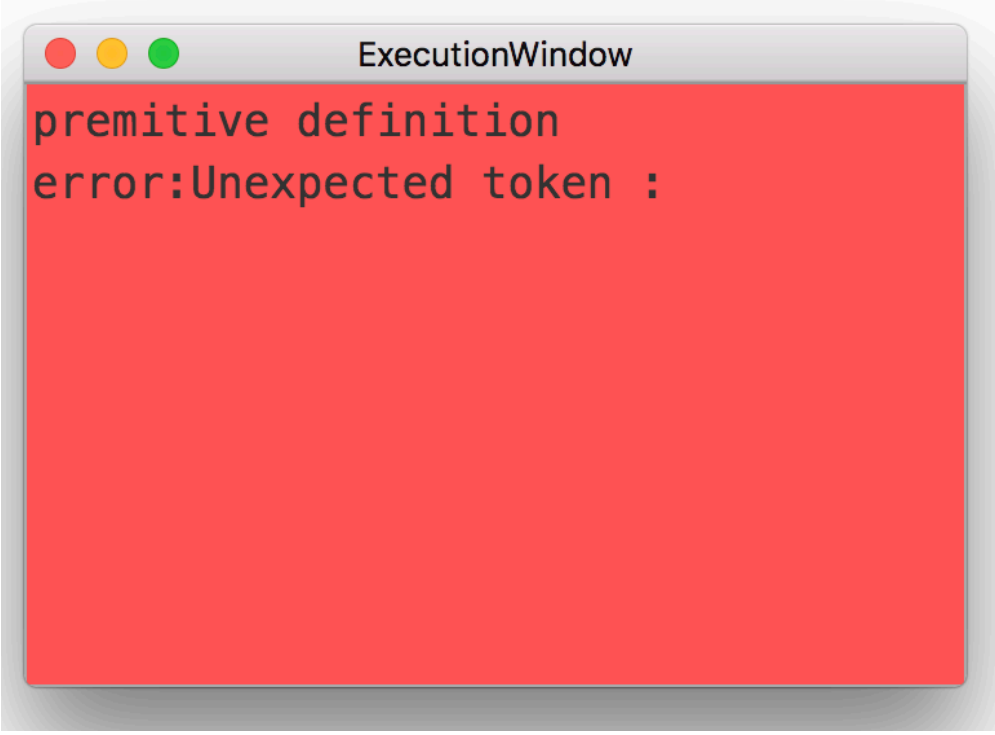

図 4.14. ExecutionWindow のエラー表示

ExecutionWindow は実行されたアプリケーションが表示されるウィンドウである．また，

ExecutionWindow は実行時に，構文解析器のパースエラーや，プリミティブ関数のエラーが起きた場合に，そのエラーを表示する．

4.6　制限

削除文字付き文字列を扱う場合，キーボードイベント列の一つである，カーソル移動履歴を用いた表現ができないという問題がある．入力や削除のような入力位置を明確に定義できるものと異なり，カーソル移動履歴は移動そのものを示すため，削除文字付き文字列上に適切に配置することができず，意味のある表現にすることができないからである．また，削除文字付き文字列を扱わず，時系列ベースのキーボードイベント列を扱うとした場合は，その実行系を作ることが難しいという問題がある．これは，既存の構文解析手法は時系列ベースにならんだ情報を解析することを想定していないためである．

第 5 章

Delevative を用いたアート制作例

この章では Delevative を用いて実際にアート制作例を示す.

5.1 無意識的にキーボードを叩く

まず無意識的キーボードを叩いた. 問題なく, キーボードの入力履歴の記録と保存を行うことができた図 5.1 に無意識にキーボードを叩いた時のコードを示す.

図 5.1. 無意識的コーディングの結果の文字列

5.2　プログラム構造を切り出す

続いて無意識的に打たれた文字からプログラム構造を切り出す作業を行った．無意識的にコードを書いていた際，最初，自分の中の無意識的な状態を汲み取るために，自身の無意識の中に入っていくような感覚を覚えていっていたので，その無意識の中に入っていく段階ごとに `enter inside` の文字列を 4 つ記述した．その後，恐怖を感じるようになったので `feel suffering` などのように，無意識的過程に含まれる無意識を一つ一つ切り出して記述した．

ここで無意識的に打たれた文字からプログラム構造を切り出す作業を行えていることを確認するべく削除時の挙動を確認した．図 5.2〜図 5.5 の無意識的プログラミングの過程を示す．図 5.2 の時点から 4 回バックスペースを押した結果の図 5.3 では，Editor 上からは文字が削除され，削除文字付き文字列上では，同様のカーソル位置から 4 つの削除文字が入力されていることがわかる．図 5.4 の時点から 5 回バックスペースを押した結果の図 5.5 では，Editor 上からは文字が削除され，削除文字付き文字列上では，末尾に削除文字が追加されている．このことから，想定したとおりに削除文字が追加されていることがわかる．

最終的なソースコードを図 5.6 に示す

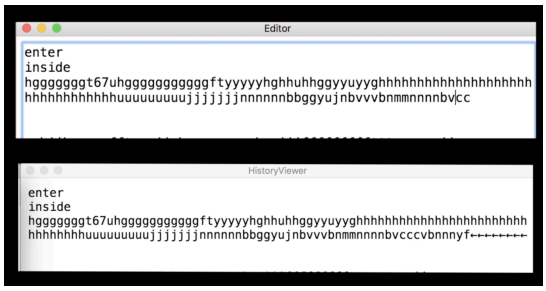

図 5.2. 入力文字の削除時の挙動 1

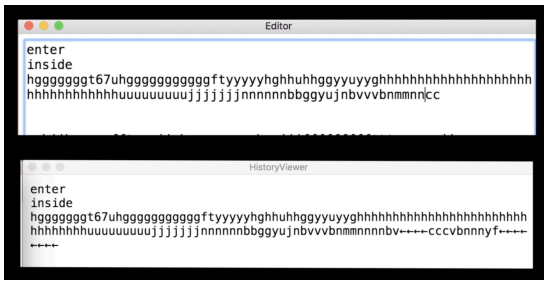

図 5.3. 入力文字の削除時の挙動 2

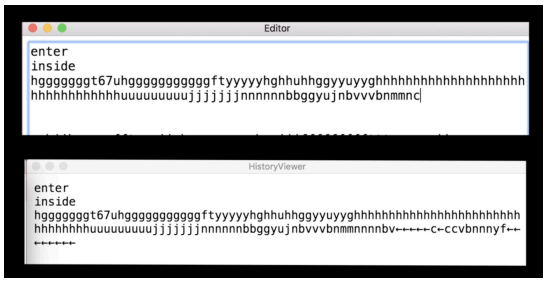

図 5.4. 入力文字の削除時の挙動 3

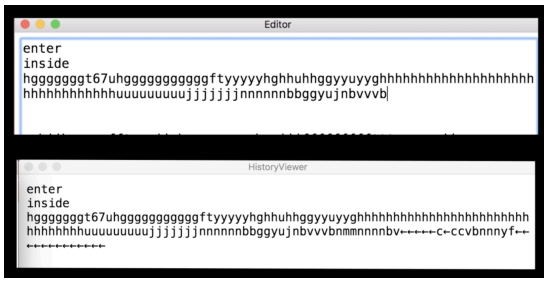

図 5.5. 入力文字の削除時の挙動 4

5.3　字句定義の記述

次に記述したソースコードを字句に分ける字句定義を記述した．図 5.7 のような字句定義になる．プログラムの形を切り出している最中に，`suffering` や `typhoon` のような関数の引数に対応する言葉の後ろに，無意識的過程が削除される形で記述したので，

```
enter inside
enter inside
enter inside
enter inside
feel suffering
freeFrom typhoon
wantToBe agressive
wantToBe free
```

図 5.6. 制作時の最終的なソースコード

`Lex.string("inside").seq(Lex.deletes())` のように後ろの削除文字付き文字列を考慮した字句定義にした.

```
var enter = Lex.string("enter ").maybe(Lex.deletes())
var feel = Lex.string("feel ").maybe(Lex.deletes())
var freeFrom = Lex.string("freeFrom ").maybe(Lex.deletes())
var wantToBe = Lex.string("wantToBe ").maybe(Lex.deletes())

var inside = Lex.string("inside").seq(Lex.deletes())
var suffering = Lex.string("suffering").seq(Lex.deletes())
var typhoon = Lex.string("typhoon").seq(Lex.deletes())
var agressive = Lex.string("agressive").seq(Lex.deletes())
var free = Lex.string("free").seq(Lex.deletes())

return [
  ["IDENTIFIER",enter.alt(feel)
    .alt(freeFrom)
    .alt(wantToBe)
    .alt(inside)
    .alt(suffering)
    .alt(typhoon)
    .alt(agressive)
    .alt(free)
],
  ["NEWLINE",Lex.string("\n")]
]
```

図 5.7. 字句定義

5.4　プリミティブ関数定義の記述

Puredata で音の実装を，Processing で映像の実装をするので，各関数の実行時に Puredata 用のアドレスと，Processing 用のアドレスのそれぞれに OSC でデータを送るようにした．実行時のタイムスタンプ情報を反映するために，ここでは，`setTimeout` を用いて最初のキー入力時からのその入力が打たれた時点の時間の差の時点で，データが送信されるようにした．ここでは入力文字に対しては，各文字コードを，バックスペースは，特定の値が Processing,Puredata に送信されるようにプリミティブ関数を定義した．

ソースコード 5.1. プリミティブ関数定義

```
function pro(add,val,time){
  setTimeout(function(){
      Lib.sendOSC(
      {
  ipAddress:"127.0.0.1",
  oscAddress:add,
  portNumber:12000,
  value:val,
      }
      )
  },time*100)
}

function pd(add,val,time){
  setTimeout(function(){
      Lib.sendOSC(
      {
  ipAddress:"127.0.0.1",
  oscAddress:add,
  portNumber:8000,
  value:val,
      }
      )
  },time*100)
}

return [
 {
    functionName:"enter",
    impl:function(history,arg){
        console.log(arg.length)
        var startTime = new Date(arg[0].histories[0].timeStamp).getMilliseconds()
        arg[0].histories.forEach((x,i) => {
            if(x.kind == "input"){
      var curTime = new Date(x.timeStamp).getMilliseconds();
              pd("/enter",-200+x.char.charCodeAt(0)*5,curTime-startTime)
      pro("/enter",x.char.charCodeAt(0),curTime-startTime)
            }
 })
    }
 },
 {
    functionName:"feel",
    impl:function(history,arg){
        var startTime = new Date(arg[0].histories[0].timeStamp).getMilliseconds()
        arg[0].histories.forEach((x,i) => {
            if(x.kind == "input"){
      var curTime = new Date(x.timeStamp).getMilliseconds();
              pd("/feel",200+x.char.charCodeAt(0)*5,curTime-startTime)
      pro("/feel",x.char.charCodeAt(0),curTime-startTime)
            }else{
              pd("/feel",2000,curTime-startTime)
      pro("/feel",255,curTime-startTime)
```

```
54                  }
55      })
56          }
57      },
58      {
59          functionName:"freeFrom",
60          impl:function(history,arg){
61              var startTime = new Date(arg[0].histories[0].timeStamp).getMilliseconds()
62              arg[0].histories.forEach((x,i) => {
63                  if(x.kind == "input"){
64              var curTime = new Date(x.timeStamp).getMilliseconds();
65                      pd("/freeFrom",200+x.char.charCodeAt(0)*5,curTime-startTime)
66              pro("/freeFrom",x.char.charCodeAt(0),curTime-startTime)
67          }
68      })
69          }
70      },
71      {
72          functionName:"wantToBe",
73          impl:function(history,arg){
74              var startTime = new Date(arg[0].histories[0].timeStamp).getMilliseconds()
75              arg[0].histories.forEach((x,i) => {
76                  if(x.kind == "input"){
77          var curTime = new Date(x.timeStamp).getMilliseconds();
78                  pd("/wantToBe",200+x.char.charCodeAt(0)*5,curTime-startTime)
79          pro("/wantToBe",x.char.charCodeAt(0),curTime-startTime)
80                  }
81      })
82          }
83      },
84  ]
```

5.5　Processing,PureData での実装

Processing[34] と Puredata[33] を用いて，実際の映像や音を表現するプログラムの実装を行った．(ソースコード 5.2, 図 5.8)

Processing は削除文字付き文字列の文字コードや関数に応じてに応じて対応する図形を描画するプログラムを実装した．また Puredata は削除文字付き文字列の文字コードと関数に応じて周波数の音を再生するプログラムを実行した．特に suffer は恐怖感を演出するために，無意識コーディング中に削除したときの音を高周波で鳴らすようにしている．

ソースコード 5.2. Processing で書いたアプリケーションのソースコード

```
1   import oscP5.*;
2   import netP5.*;
3
4   OscP5 oscP5;
5   int enter = 0;
6   int feel = 0;
7   int freeFrom = 0;
8   int wantToBe = 0;
```

```
9
10  void setup(){
11      size(1200,800);
12      frameRate(60);
13
14      oscP5 = new OscP5(this,12000);
15  }
16
17  void draw(){
18      background(0);
19      colorMode(RGB,256);
20      noStroke();
21
22      fill(feel,feel,feel);
23      rect(0,0,400,1200);
24      fill(enter,enter,enter);
25      rect(400,0,400,1200);
26      fill(freeFrom,freeFrom,freeFrom);
27      rect(800,0,400,600);
28      fill(wantToBe,wantToBe,wantToBe);
29      rect(800,600,400,600);
30  }
31
32  void oscEvent(OscMessage theOscMessage) {
33    if(theOscMessage.checkAddrPattern("/enter") == true){
34        enter = theOscMessage.get(0).intValue();
35    }
36    if(theOscMessage.checkAddrPattern("/feel") == true){
37        feel = theOscMessage.get(0).intValue();
38    }
39    if(theOscMessage.checkAddrPattern("/freeFrom") == true){
40        freeFrom = theOscMessage.get(0).intValue();
41    }
42    if(theOscMessage.checkAddrPattern("/wantToBe") == true){
43        wantToBe = theOscMessage.get(0).intValue();
44    }
45  }
```

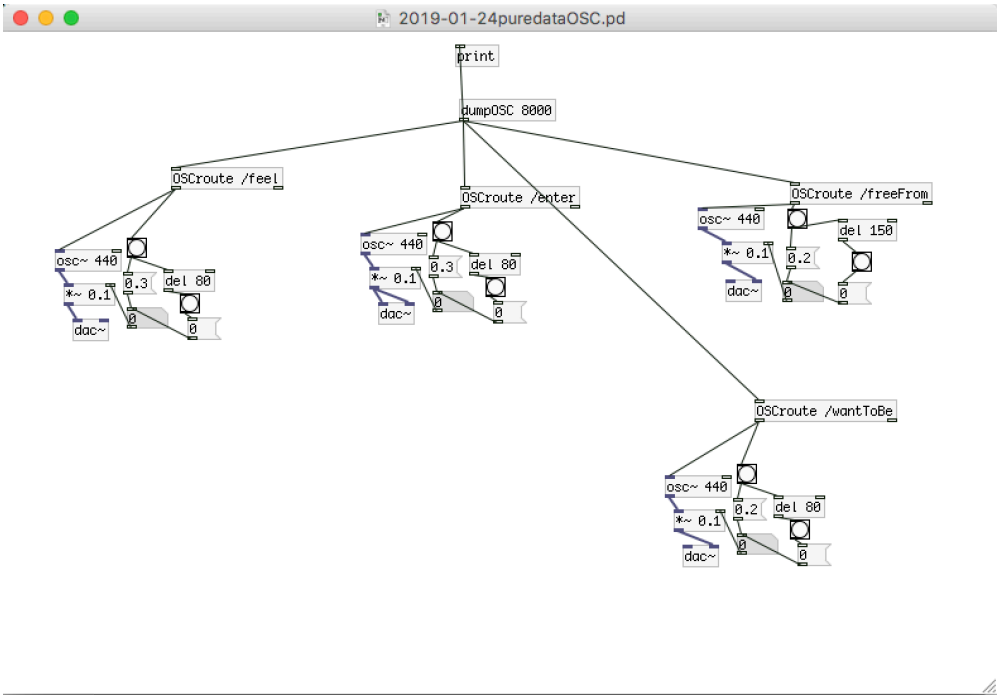

図 5.8. Puredata のプログラム

5.5.1　実行結果

最終的な実行結果は，各関数ごとのキータイプ時の文字や削除文字に対応する音を打鍵時刻に応じて再生しつつ，各感情に対応する場所に図形を描画していくことで，自身の無意識を表現した．`https://youtu.be/ukPWtlnv1mM` で実行結果の様子を見ることができる．

第 6 章

まとめと課題

本研究では，無意識を表現するためにはマクロかつミクロな条件を満たしたアートであることが理想であると述べた上で，無意識を表現するためのマクロかつミクロな条件を満たしたアートである無意識的プログラミングを提案した．無意識的プログラミングは，無意識的にプログラムを記述することで，無意識的にキータイプを行い，無意識的な動作の記録としてキーイベント配列の形で保存を行った後に，構文解析器のカスタマイズでキーイベント配列から抽象構文木を作り出した後に，実行器のカスタマイズを行うことで，実際のアプリケーションの実行結果を定義し，最後にキーイベント配列ソースコードをカスタマイズした構文解析器，実行器で実行するアートである．

一般的なキーイベント配列は時系列に並んだ情報であるため，既存の構文解析手法で解析することができないという問題があったが，本研究で開発した統合実行環境 Delevative では，キーイベント配列を入力文字と削除文字がメタ情報付きで，カーソル位置を適用した順番で記録される削除文字付き文字列という形で記録することによって，既存の構文解析手法を用いた構文解析を行うように実装した．

Delevative を用いた無意識的プログラミングにおけるミクロな表現は無意識的なキータイプを行うことで実現し，マクロな表現は，まず無意識的な痕跡をプログラム構造に書き換えた後に，構文解析器のうち，字句定義のみをアーティストがカスタマイズし，その後，実行器のうち，プリミティブ関数のみをカスタマイズすることによって実現した．

Delevative の実装においては，削除文字付き文字列を用いた字句解析器のカスタマイズのために，PackratParsing を用いたパーサーコンビネータと，ユーザーが PEG を用いて字句定義を行うための DSL を実装し，キーイベントから削除文字付き文字列を適切に生成するために，通常の文字列と削除文字付き文字列を計算し，それを元に Editor 上のカーソル位置から適切な位置にキーイベントを挿入できるように実装した．また，プリミティブ関数定義時に `OSC` ライブラリを利用できるようにし，外部アプリケーションと通信できることにしたことで，Delevative を用いたアート制作を現実的に行えるようにした．

削除文字付き文字列を用いると，カーソルキーの打鍵がキータイプとしてキーイベント配列上に保存されないという問題があるため，構文解析の段階で記録するなど，カーソルキーの打鍵を加えることが本質的な今後の研究課題である．

発表文献と研究活動

(1) 佐藤輝年．千葉滋．人間的動作を活かしたライブコーディングのためのソフトウェアシステム．PPL2018, 2018.03.05-07.(ポスター発表)
(2) 佐藤輝年，山崎徹郎．千葉滋．アート体験のためにプログラミング過程の情報を構造化する上でのトークナイズ手法．日本ソフトウェア科学会第 35 回大会, 2018.08.28-31.(論文発表, 口頭発表)
(3) 佐藤輝年．アクションペインティングスタイルプログラミングのためのプログラミングシステム．SIGPX, 2018.12.03.(口頭発表)

参考文献

[1] Harold Rosenberg. The american action painters. *ART News*, 1952.

[2] Jason Edward Kaufman. Painting: What the mind's eye sees: Action painters were postwar exemplars of american individualism. *The American Scholar*, Vol. 77, No. 2, pp. 113–117, 2008.

[3] Clement Greenberg. American-type'painting, art and culture, 1961.

[4] 澁澤龍彦. 澁澤龍彦 西欧芸術論集成 下 (河出文庫). 河出書房新社, 4 2010.

[5] ラルフ・シーブラー. ダリ - - 夢のリアリティー (岩波アート・ライブラリー). 岩波書店, 9 2010.

[6] クリストファーマスターズ. ダリ (アート・ライブラリー). 西村書店, 新装, 1 2012.

[7] 井上靖. 世界の名画 (22) エルンストとダリ - シュルレアリスムの絵画. 中央公論社, 〔新装カンヴァス版〕, 10 1994.

[8] Nick Collins, Alex McLean, Julian Rohrhuber, and Adrian Ward. Live coding in laptop performance. *Organised sound*, Vol. 8, No. 3, pp. 321–330, 2003.

[9] 田所淳. 演奏するプログラミング、ライブコーディングの思想と実践 - Show Us Your Screens. ビー・エヌ・エヌ新社, 12 2018.

[10] Sang Won Lee and Georg Essl. Live writing: Asynchronous playback of live coding and writing. In *Proceedings of the International Conference on Live Coding*, 2015.

[11] David Ogborn, Jamie Beverley, Luis Navarro del Angel, Eldad Tsabary, and Alex McLean. Estuary: Browser-based collaborative projectional live coding of musical patterns. In *International Conference on Live Coding (ICLC) 2017*, 2017.

[12] Alex McLean. Making programming languages to dance to: Live coding with tidal. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling Design*, pp. 63–70, 2014.

[13] Kasibatla Saketh and Warth Alessandro. Seymour: Live programming for the classroom. In *SPLASH LIVE 2017*, 2017.

[14] Tomoki Imai, Hidehiko Masuhara, and Tomoyuki Aotani. Shiranui: a live programming with support for unit testing. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, pp. 36–37. ACM, 2015.

[15] Lauren McCarthy, Casey Reas, and Ben Fry. *Getting Started with p5.js: Making Interactive Graphics in JavaScript and Processing (Make: Technology on Your Time)*. Maker Media, Inc, 1 edition, 10 2015.

[16] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pp. 51–64. ACM, 2002.

[17] Conal Elliott and Paul Hudak. Functional reactive animation. In *ACM SIGPLAN Notices*, Vol. 32, pp. 263–273. ACM, 1997.

[18] Leo A Meyerovich, Arjun Guha, Jacob Baskin, Gregory H Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for ajax applications. In *ACM SIGPLAN Notices*, Vol. 44, pp. 1–20. ACM, 2009.

[19] Hugo Liu and Henry Lieberman. Metafor: Visualizing stories as code. In *Proceedings of the 10th International Conference on Intelligent User Interfaces*, IUI '05, pp. 305–307, New York, NY, USA, 2005. ACM.

[20] Hugo Liu. Montylingua v2.1. `http://alumni.media.mit.edu/~hugo/montylingua/`, 2004.

[21] 星野大樹, 林晋平, 佐伯元司. ソースコード編集操作の自動グループ化. コンピュータ ソフトウェア, Vol. 31, No. 3, pp. 277–283, 2014.

[22] 林晋平, 大森隆行, 善明晃由, 丸山勝久, 佐伯元司. ソースコード編集履歴のリファクタリング手法. ソフトウェア工学の基礎 XVIII — 第 18 回ソフトウェア工学の基礎ワークショップ予稿集, pp. 61–70. nov 2011.

[23] 大森隆行, 丸山勝久. 開発者による編集操作に基づくソースコード変更抽出. 情報処理学会論文誌, Vol. 49, No. 7, pp. 2349–2359, 2008.

[24] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pp. 313–324, New York, NY, USA, 2014. ACM.

[25] 加藤淳, 五十嵐健夫. Pressing: 打鍵の強さで出力が変わるビジュアルインタプリタ. インタラクション 2010, インタラクション 2010 デモ, pp. 191–192. Information Processing Society of Japan (IPSJ), Mar 2010.

[26] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. *SIGPLAN Not.*, Vol. 39, No. 1, pp. 111–122, January 2004.

[27] Electron:build cross platform desktop apps with javascript, html, and css. `https://electronjs.org/`, (2019.01.15 アクセス).

[28] Chromium - the chromium projects. `https://www.chromium.org/Home`, (2019.01.15 アクセス).

[29] Matthew Wright, Adrian Freed, et al. Open soundcontrol: A new protocol for com-

municating with sound synthesizers. In *ICMC*, 1997.

[30] node-osc - npm. https://www.npmjs.com/package/node-osc, (2019.01.15 アクセス).

[31] vvvv - a multipurpose toolkit. https://vvvv.org/, (2019.01.15 アクセス).

[32] Max software tools for media — cycling'74. https://cycling74.com/products/max/, (2019.01.15 アクセス).

[33] Pure data — pd community site. https://puredata.info/, (2019.01.15 アクセス).

[34] Processing.org. https://processing.org/, (2019.01.15 アクセス).

[35] openframeworks. https://openframeworks.cc/, (2019.01.15 アクセス).

謝辞

本研究を進めるにあたり，研究方針や論文構成，発表手法，研究アイディアなど多岐にわたって手厚くご指導賜りました指導教員の千葉滋教授に心より感謝致します．研究を進める過程で多くの困難があった中，とても大きく強い支えとなってとなってくださいました．構文解析手法に関するアドバイスとサポート，論文の執筆や研究の発表に関する多くのご指導を頂きました山崎徹郎氏に厚くお礼を申し上げます．穐山空道助教には，急なお願いであったにもかかわらず，論文と発表について時間をかけてご指導いただきました．大変感謝しております．最後に，日頃から研究室生活やミーティングを通じてご支援くださった千葉研究室の皆様に心より感謝致します．