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

佐藤 輝年　山崎 徹郎　千葉 滋

アクションペインティングのように，作品を作る過程そのものを重視するアート思想に基づきプログラミングをすることができれば，一つのアート体験として成り立つことが考えられる．こうしたプログラミングによるアート体験（アクションコーディング）のためのソフトウェアを作るためには，プログラミング過程の情報を構造化し，過程情報を活用しながら実行するアーキテクチャが必要である．本研究では，こうした背景を踏まえプログラミング過程の情報としてエディターの入力履歴を用いた場合に，入力履歴を構造化する上で必要な字句解析手法を提案する．一般的な言語では字句解析器は正規文法で定義することができるが，入力履歴を扱う場合は，字句解析の段階で元情報を階層的に解釈する必要があるため，字句解析器は正規文法では定義することができないという課題が明らかになった．発表内ではこうした入力過程特有の課題やその解決策を詳細に解説する．

1　はじめに

自身の無意識を具現化するようなアートは数多い．例えば，アクションペインティング[3] という絵画を描く技法がある．この技法によって描かれた絵画は一見すると線や斑点が無作為に描かれたように見える．なぜならば，アクションペインティングでは，キャンバスは何かを描くための道具ではなく，キャンバスを素材との無意識的な格闘の場として捉えられており，アクションペインターはその思想のもと，キャンバスに筆を走らせているからである．

我々はこのような無意識を具現化するようなアートをプログラミングで実現するべく，そのためのプログラミングシステムを開発している．本論文ではそのようなアートを実現するプログラミングシステムの概要と，その中から特に字句解析部の設計について述べる．アートを制作するために，プログラマ (アーティスト) は 3 つのステップでこのプログラミングシステムを利用する．まず，無意識的コーディングと無意識の解釈のためのコーディングをし，次にそのテキストに対して意味を与えるように字句解析方法とプリミティブ関数を定義し，最後に記述したプログラムを実行する．無意識的なコーディングによって，プログラマ (アーティスト) は無意識を解釈するための素材となるプログラム風のテキストを作り出し，無意識の解釈のためのコーディングで，プログラムに対する自身の解釈を記述する．その後，テキストに対して意味を与えるような字句解析方法とプリミティブ関数を設定することで，自身の無意識を構造的解釈を明示的に行い，その構造に具体的な意味を与える．最後に，プログラムを実行し，自身で鑑賞することで，無意識の解釈，意味付けに違和感がないかを確かめる．この一連の流れによってプログラマ (アーティスト) は単に無意識を反映するだけでなく，無意識の解釈，具現化にまで及ぶアート制作を行うことができる．

このプログラミングシステムのアーキテクチャは，履歴付きプログラムを保存するエディタ，履歴付きプログラムのための字句解析器，構文解析器，履歴情報を利用するプリミティブを定義できる実行器から構成される．その中で，本論文では特に字句解析部の設計について述べる．なぜならば，無意識を解釈し，具現化する上で，過程を辿ることが重要だからである．

Tokenizing method for structuring information of programming process for art experience

Akitoshi Sato, Yamazaki, Tetsuro, Chiba Shigeru , 東京大学, The University of Tokyo.

提案する字句解析手法では，ソースコードを書くに至るまでの過程を用いるために，入力として普通の文字列のソースコードではなく，キー入力履歴を用いてトークンを分割する．どうやってトークンを分割するのが望ましいかはアーティストによって異なると考えられるので，字句はユーザーがそれぞれ定義する．そのため我々のプログラミングシステムは，PEG [1] [2] として記述された字句の定義から字句解析器を生成する．字句の定義は一般的に正規文法として記述されるが，キー入力履歴に含まれる文字削除と削除される文字とを対応付けるにはネスト構造を扱うことができない正規文法では能力不足である．

scannerless parsing [4] のテクニックを用いることで、字句解析なしに構文解析することも可能である。しかし、我々の設計するプログラミングシステムではユーザーごとに異なる方法でプログラムを解釈する。そのため、字句解析器と構文解析器は別々のモジュールとして提供される方が、字句解析器単体でのカスタマイズが可能であり、扱いやすい。

以下，２章では字句解析部に求められる機能を説明し，３章ではその機能を実現する字句解析部の設計について述べる．４章は，その字句解析部を含む我々が開発中のプログラミングシステムの概要を示す．５章はまとめである．

2 無意識を具現化するアートのための字句解析

無意識を具現化するアートのためのプログラミングシステムでは，プログラマ（アーティスト）がプログラムをタイプするとき，タイプされた文字だけでなく，そのタイミングや文字削除の様子の履歴についても記録する．現在のところ，カーソルの移動履歴は記録しない．

本システムで入力されたプログラムは，後退文字 (back space, 以下 ← と表記) を含む文字列であり，各文字にはタイプされた時刻が属性として付加される．したがってプログラムは例えば次のような文字列となる．

```
prin h←←t hello
```

後退文字 ← が 2 回タイプされているので，最終的にエディタ上に見えるプログラムは下のようになるが，本システムでは上のように記録する．

```
print hello
```

このプログラムは通常の場合と同様に，まず字句解析器によって字句 (token) に分解される．字句に含まれない空白やコメントは存在しないものとして無視される．字句解析の際，本システムではプログラムをタイプした過程も残したいので，例のプログラムの場合，例えば「`prin h←←t`」と「`hello`」の 2 つの字句に分解しなければならない．

一般に，字句解析器には字句の定義を正規表現で与えるが，本システムの場合，後退文字も考慮しなければならないので正規表現ではうまくいかない．例えば次のような定義なら一見うまくいきそうだが，

```
identifier: [A-Z_a-z][A-Z_a-z0-9←]*
```

と 2 文字目以降は後退文字でもよいことにすると，上の例は「`prin`」と「`h←←t`」と「`hello`」の 3 つの字句に分解されてしまう．後退文字とそれによって削除された文字の対応関係を考慮して字句解析しなければならないが，正規表現ではそのような対応関係を扱えないからである．

どのような字句に分解するかには，複数のやり方があり，プログラマ（アーティスト）が自身で定義できるようにしたい．例えば

```
print hell
```

までタイプしてから，`print` と `hell` の間の空白文字まで合わせて 5 文字削除し

```
print hi
```

と改めて空白文字から 3 文字タイプした場合，つまりプログラムが

```
print hell←←←←← hi
```

であった場合，

```
print hell←←←←←
hi
```

と分解したいかもしれないが，

```
print
 hell←←←←← hi
```

と分解したいかもしれない．プログラマが望ましい定義を本システムに与えられるように，個々の字句の定義を表現する方法が必要である．

3 PEG を用いた字句解析

後退文字を含むプログラムをプログラマ（アーティスト）が望むように字句解析するため，我々が開発しているプログラミングシステムには PEG (Parsing Expression Grammar) [2] で表現した字句の定義を与えることができる．我々のシステムは与えられた字句の定義にしたがって，プログラムを packrat パーサ [1] で解析し，字句に分解する．通常，構文解析に用いる手法を字句解析に適用することで，後退文字を含むプログラムを字句解析する．後退文字とそれによって削除される文字の関係は，括弧の対応関係と同様で，また入れ子構造をとる．入れ子になる対応関係を表現できる表現法を用い，それを解析できる解析法を用いることで，本システムに必要な字句解析をおこなう．

PEG を用いれば，前章の例は次のように字句 `identifier` を定義できる．

```
deleted := any deleted bs / any bs
identifeir := letter (deleted / letter)*
              (deleted / sp)*
```

ここで `any` は任意の文字と一致する非終端記号，`bs` は後退文字 ← と一致する非終端記号，`letter` は `A-z` と一致する非終端記号，`sp` は空白文字と一致する非終端記号である．`*` は 0 回以上の繰り返しを表す．記号 `/` は ordered choice であるので，まず `/` の左辺を試し，一致しない場合だけ右辺を試す．このため後退文字が含まれている限り，そちらが優先的に一致する．この定義であれば，次のプログラム

```
print hell←←←←← hi
```

は，次のような 2 つの字句に分解される．

```
print hell←←←←←
hi
```

一方，字句の定義を変えて

```
deleted := any deleted bs / any bs
identifeir := (deleted / sp)*
              letter (deleted* letter)*
```

とすれば次のような 2 つの字句に分解される．

```
print
 hell←←←←← hi
```

プログラマ（アーティスト）は，システムに与える字句の定義をこのように変えることで，自身が望む表現が得られるように字句解析を実行させることが可能である．

4 無意識を具現化するアートのためのプログラミングシステム

我々は前章で示した字句解析器を含むプログラミングシステムを現在開発中である．現在実装中のプログラミングシステムの構成を図 1 に示す．このシステムは履歴付きプログラムを保存するエディタ，履歴付きプログラムのための字句解析器，構文解析器，履歴情報を用いたプリミティブを定義できる実行器からなる．これらの各モジュールは Electron を用いて実装されている．システム内には標準で備わっている言語があり，これをカスタマイズすることで利用する．標準で備わっている言語は，一般的な言語の標準的なものに近い機能が備わっている．具体的には条件分岐 (`if`)，ループ (`while`)，変数の定義，プリミティブ関数の呼び出しが行えるが，関数定義を行うことはできない．

本システムによって作られたプログラマ (アーティスト) の作品は，無意識的な過程そのものを重視したものになる．そのため，プログラムの最中そのものが鑑賞の対象であると考えて，プログラムを書いている最中の様子の映像や，作品の解説のためのキャプションとともに，プログラムを展示しても良いし，観客にあえてその過程の推察を求めるようなアートの場合は，最終的な実行されたプログラムのみを展示しても良い．あるいは，自身の無意識の解釈・具現化そのものが目的であればアートを制作する体験そのものを楽しみ，展示をしないという選択肢もあり得る．

以下，プログラマ（アーティスト）が本プログラミングシステムを利用するための 3 つのステップについて述べる．

4.1 プログラミング言語を用いたコーディング

まず，なにを書くかを意識せずに，無意識的なコーディングを行う．プログラマ (アーティスト) が本システムを用いてアートを制作する目的は，思考の裏側にある無意識を解釈し具現化することにある．そのため，自分の無意識を解釈可能な形に媒体化する必要がある．無意識的な行為には自身の無意識が反映されるので，無意識的なコーディングをすることによって，その媒体として，プログラム風テキストを作り出す．

例えば，プログラマ（アーティスト）はプログラム 1 のような一見無意味に見えるようなソースコードを無意識的に記述する．

プログラム 1
```
helhasdkfasdkjasdkj
dfjkasdfjadjfp
```

次に，プログラマ（アーティスト）は無意識的に記述したソースコードの解釈の結果やその表現方法を考察した上で，ソースコードの編集を行う．これは，媒体化された無意識を明示的に解釈する工程である．プログラミングを手段に用いると，ソースコードを解釈可能な形式で記述するということが自動的に求められるので，自然な形で無意識の解釈をアーティストに促すことができる．

例えば，ここではプログラマ（アーティスト）が自身のソースコードを誰かに助けを求めるような気持ちをたどりながら書いたと解釈し，その気持ちを音で表現したいと考えた場合，プログラム 2 のように，その結果を反映して，意味のない羅列から不要な文字列を削除し，「help」というテキストにした上で，プログラム 3 のように，help を引数に取る関数 voice を呼び出すソースコードを記述する．

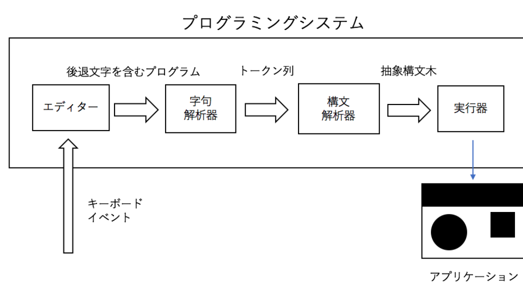

図 1

プログラム 2
```
helha..←..←p
```

プログラム 3
```
voice "helha..←..←p"
```

4.2 字句解析器とプリミティブ関数のカスタマイズ

4.1 章のステップ後に字句解析器とプリミティブ関数のカスタマイズを行う．まずは，字句解析器のカスタマイズを行う．字句解析器のカスタマイズを行うことで，履歴がどのトークンに属するかを形式的に記述することになるので，プログラマ (アーティスト) により詳細な解釈を促すことができる．例として，以下の 3 つのトークンを標準で備えられた言語に定義する．本来は数字をトークンとして扱う事もできるが，今回は数値を表すトークンが存在しないので，identifier と string のみを定義する．dq はダブルクォーテーションを表す．

```
deleted := any deleted bs / any bs
identifeir := letter (deleted / letter)*
              sp*
string := dq (deleted / sp)*
              letter (deleted* letter)* dq
```

今回のプログラムでは影響はないが，string トークンにできるだけ履歴情報が集まるようにトークン規則を定義した．これは無意識的な過程には感情が表れていると解釈し，感情を表すトークン"helha..←..←p"に込めたいという意図に基づいている．今回，ソースコード上では，関数呼び出しのみが行われているので，プログラムの実行の際には構文解析器からプリミティブ関数の実行を表す call ノードが出力される．call ノードは関数名を表す子ノードである name と引数を表す子ノード arg1,arg2... を持つ．

次に，プリミティブ関数を実行器の一部である Premitive.ts を書き換え，ソース内の primitive オブジェクトのプロパティに追加することで定義す

る．この工程によって，無意識的に解釈した結果を具現化することができる．

Primitive.ts
```
const premitive = {
  voice:function(ownHistory:History[],
        arg:ParamWithHistories<string>)
   {
    var inputs = arg.histories
         .filter(x =>
           x.kind == HistoryKind.Input
         )
    for(var i=0;i<arg.length;i++){
       sound(arg.histories[i].inputedChar
            ,arg.histories[i].timeStamp)
    }
   },
}
```

ソースコードの編集と字句解析器のカスタマイズは理性的な工程であるが故に，その解釈に対して感情的な想いを込めることは難しい．ここでは理性的な解釈に対して画像や音といった直感的な解釈を促すアプリケーションを実装することで，無意識の解釈に対して感情的側面を反映する．

プリミティブ関数は第一引数として，呼び出し側の call ノード全体の履歴情報を取り，第二引数以降は `call` ノードの `arg1,arg2...` の値と履歴情報を取得できる．上記の `Premitive.ts` では string 型の引数を 1 つ取る関数 `voice` を定義している．引数の型で用いられている History,ParamWithHistories<T> はプログラミングシステム側が提供する．通常のプログラミング言語では関数で定義できる様々な映像や音を出力するような関数も，プリミティブ関数として定義する．例えば上記の `Primitive.ts` ではコード 3 で `helha..` ←`..` ← `p` を引数に取る関数 `voice` を定義した．`voice` は，途中過程で入力された文字の数だけの種類の声を，それぞれの打たれた時間に応じた周波数で発声するような関数であり，今回のプログラムでは `helha..` ←`..` ← `p` が引数に取られる．`helha..` ←`..` ← `p` という感情を表すトークンに紐付いた履歴から様々な音声で一度に発声することで，自身の感情を表現するという意図がある．`voice` を定義する際に，Typescript 上で別途定義した第一引数の文字列を第二引数の周波数で発声するような関数 `sound(text,freq)` を用いている．

4.3 プログラムの実行

4.2 章のステップ後にエディターの実行ボタンを介して，プログラムの実行を行う．プログラムを実行し，自身で鑑賞することで，無意識の解釈，意味付けに違和感がないかを確かめる．上で定義したプリミティブを用いて，プログラム 3 を実行した場合は，プリミティブで定義されている関数 `voice` が実行され，プログラム 1 で書かれた文字の数の声がそれぞれ別々の周波数で一度に鳴るようなアプリケーションが実行される．

5 まとめ

本論文では無意識を具現化するアートのためのプログラミングシステムの概要を利用方法とアーキテクチャの面から説明した上で，そのようなプログラミングシステムを作るための字句解析手法を示した．PEG を字句解析器に用いることで，入れ子構造を表現ができ，履歴情報を含むプログラムの字句解析をすることができる．今後の課題は開発中のプログラミングシステムの開発を続け完成させること，理想的なアートのために標準で備えられた言語をカスタマイズするのではなくユーザー定義言語がデザインできるような機能を実装することである．

参 考 文 献

[1] Ford, B.: Packrat Parsing:: Simple, Powerful, Lazy, Linear Time, Functional Pearl, *SIGPLAN Not.*, Vol. 37, No. 9(2002), pp. 36–47.

[2] Ford, B.: Parsing Expression Grammars: A Recognition-based Syntactic Foundation, *SIGPLAN Not.*, Vol. 39, No. 1(2004), pp. 111–122.

[3] Rosenberg, H.: The American Action Painters, *ART News*, (1952).

[4] Salomon, D. J. and Cormack, G. V.: Scannerless NSLR(1) Parsing of Programming Languages, *SIGPLAN Not.*, Vol. 24, No. 7(1989), pp. 170–178.