プログラミング教育のための視覚的ドメイン特化言語
Domain-Specific Visual Languages for Programming Education

栗原あずさ
Azusa Kurihara
法政大学大学院情報科学研究科
Email: azusa.kurihara.6s@stu.hosei.ac.jp

Abstract

Visual block-based programming is useful for various users such as novice programmers because it provides easy operations and improves the readability of programs. Also, in programming education, it is known to be effective to initially present basic language features and then gradually make more advanced features available. However, the cost of implementing such visual block-based languages remains a challenge. In this paper, we present a programming environment for providing visual block-based domain-specific languages (visual DSLs) that are translatable into various programming languages. In our environment, programs are built by combining visual blocks expressed in a natural language. Blocks represent program elements such as operations and variables. Tips represent snippets, and macro blocks represent procedures. Using Tips and macros makes code more abstract, and reduces the number of blocks in code. Visual DSLs can be a front-end for various languages. It can be easily restricted and extended by adding and deleting blocks. We applied our programming environment to Processing, an educational programming language for media art. We show that the environment is useful for novice programmers who learn basic concepts of programming and the features of Processing.

1. はじめに

プログラミング言語の作成と習得にはコストがかかる．特にプログラミング教育用の言語を作成する際には，文法の設計，開発環境，ヘルプやリファレンス，課題などを作成する必要がある．課題の目的は順次実行・分岐・反復といったプログラミングのコンセプトおよび言語機能の理解である．課題においては基本的な機能から段階的に発展していくことで学習効率が上がることが知られている．従って，言語機能の制限・拡張が容易な言語および初学者が容易にコードを読み書きできる環境が必要とされている．

プログラミング言語の三つの規約，文法・意味論・語用論について Lloyd Allison [1] は以下のように述べる．

> The complete definition of a programming language is divided into *syntax*, *semantics* and *pragmatics*. Syntax defines the structure of legal sentences in the language. Semantics gives the meaning of these sentences. Pragmatics covers the use of an implementation of a language ....

本論文ではプログラミング教育のための視覚的ドメイン特化言語（Domain Specific Language，以下 DSL）とその開発環境について述べる．視覚的 DSL はテキストベースの言語に変換可能なブロックを用いてコードを記述する．変換される言語を以下ホスト言語と呼ぶ．ブロックは自然言語でそのはたらきを示し，文法エラーのないコードに変換される．ユーザはブロック同士をドラッグ&ドロップで組み合わせることで記述するためキーボードに慣れていない初学者でも容易に記述できる．

本開発環境は web アプリケーションとして提供され，HTML，JavaScript，CSS で記述されている．視覚的 DSL の開発環境は，ブロック，エディタおよびトランスレータを含む．更に，開発環境にホスト言語の実行系を組み込むことで，ライブコーディング [2] が可能になる．これにより，上記三つの規約の理解について以下のように解決を図る．

1) ブロックを文法エラーのないコードに変換する．
2) ブロックのはたらきを平易な自然言語で表現する．
3) スニペットを表す「tips」ブロックを定義する．

プログラミング初学者は，視覚的 DSL を用いることでプログラミングのコンセプトとホスト言語の機能を学ぶことができる．また，変換されたコードとブロックを比較することでテキストベースの言語への移行を支援する．

視覚的 DSL の実装例として Processing [3] のアプリケーション開発のための視覚的 DSL について述べる．Processing は Java をベースとするメディアアートのためのプログラミング言語であり，簡潔な表現で視覚的な結果を得られることからプログラミング教育に利用されることが多い．プログラミング初学者が Processing を学ぶ際には，文法やライブラリ，データ構造を理解しにくい．そこでブロックを選択的に提供することで視覚的 DSL の機能を拡張・制限する．Processing のための視覚的 DSL は，ブロック，エディタ，トランスレータ，Processing の実行系を持ち，テキストコードと実行結果を表示する．本環境はライブコーディングをサポートし，ブロックに変更があるたび変換と Processing の実行が自動で行われる．

また，学習教材の例として，プログラミング初学者および Processing 初心者のための教材を作成した．教材の目的は，プログラミングのコンセプトおよび Processing の機能を学ぶことである．Processing の機能を学ぶ際には，はじめに直線を書くなど基本的な機能を学び，徐々に手続きなどの高度な機能を学ぶ．Processing を通じて汎用的なプログラミングのコンセプトを理解することで，テキストベースの言語への移行をサポートする．

2. 関連研究

最も古い教育用プログラミング言語の一つに Logo [4] が挙げられる．Logo はタートルグラフィクスをサポートし，タートルと呼ばれる図を動かし線を描画する．しかし，そのコーディングには以下の問題がある．

Supervisor: Prof. Hiroshi Hosobe

1) ユーザの多くがキーボードの操作に慣れていない.
2) 命令や引数が英語表記であるため読み書きしにくい.
3) 変数とポインタは初学者には理解しにくい.

そこで，ブロックベースのプログラミング言語である Scratch [5]，Blockly [6]，Waterbear [7]，Tiled Grace [8] などが開発された．ブロックはドラッグ&ドロップなどで簡単に操作でき，そのはたらきは平易な自然言語で表現されているため，プログラミング初学者に広く使われている．

Scratch はブロックベースの言語とその開発環境であり，子供だけでなく幅広い年齢層のユーザに利用されている．Scratch はタートルグラフィクスをサポートし，リストなどのデータ構造，ブロックの作成などの高度な機能を備えている．作成するブロックの形状は一種類に限られ，また既存のブロックの組合せでそのはたらきを定義するが，Scratch 上に存在しない機能を追加できない．また，Scratch は Flash で実装されていることから，テキストコードなどのアウトプットを得ることができない．

Blockly はブロックのトランスレータを実装しており，JavaScript，Python，Dart，XML 形式のテキストコードを得られる．Blockly は迷路など様々な種類のパズルを実装しており，これを解くことで段階的に機能を学ぶことができる．Blockly で作成するブロックは Scratch より形状を細かく指定できるが，ブロックのはたらきはテキストコードで記述するためホスト言語の理解を要する．従って，初学者にはブロックを作成できない．

テキストベースの教育用プログラミング言語とその環境には Haskell Platform などが挙げられる．Haskell Platform は「Batteries included」，すなわち実行に必要な環境とライブラリが全て含まれており，ダウンロード後すぐに利用できる．また，Try Haskell [9] は Haskell の基本的な機能のみを実装した web チュートリアルである．これによって，ユーザは基礎的な文法や機能を学ぶことができる．

3. 視覚的 DSL の設計

3.1. 内部 DSL と外部 DSL

ドメイン特化言語の対義語は汎用言語（General-Purpose Language，以下 GPL）である．GPL には C 言語や Java などが挙げられる．GPL は様々な分野の問題解決に利用され，その多くがチューリング完全である．対して，DSL は限られた分野に対して用いられ，その分野に特化した機能を多く備えている．一般に GPL と DSL では DSL のコードの方が高い生産性・可読性・再利用性を持つ．

DSL には内部 DSL と外部 DSL がある．内部 DSL はホスト言語の API やライブラリとして実装され，ホスト言語で記述される．Processing は Java の，Processing.js は JavaScript の内部 DSL である．外部 DSL はホスト言語のコードを生成し，ホスト言語と異なる言語で記述されることもある．Blockly は JavaScript などの外部 DSL である．外部 DSL を用いることでホスト言語より短く抽象的・意味的な表現が可能にし，生産性を高める．

ホスト言語の実行系を視覚的 DSL の環境に組み込むことで変換と実行を自動で行うライブコーディングを可能にする．例えば，Processing のための視覚的 DSL は外部 DSL としてブロックをホスト言語に変換し， JavaScript の内部 DSL である Processing.js が実行する．このように，ブロックの持つ機能を最小限に抑えコードの記述と実行を分離することで，視覚的 DSL の様々な言語への適用を容易にする．

3.2. 視覚的 DSL のインタフェース

本節では Processing のための視覚的 DSL の外観（図 1）について述べる．視覚的 DSL の画面は，パレット，ワークスペース，キャンバスとテキストコードに分割される．

パレットは利用可能なブロックを表示する．ブロックはそのはたらきごとにブロックセットに分類される．ユーザはパレットを見ることでブロックとそのはたらきを確認できるため，リファレンスとしても利用できる．

ユーザはワークスペース上でプログラムを編集する．パレットからドラッグ&ドロップしたブロックを，ワークスペース上で組み合わせたり切り離したりすることによってプログラミングを行う．また，ワークスペースではパイメニューを展開できる．パイメニューはブロックセットごとに階層的に表示され，選択したブロックはワークスペースに追加される．これによりカーソルの移動距離が減少し，より効率的なコーディングが可能になる．

実行結果のポップアップには，Processing のキャンバスとテキストコードが表示される．ブロックにカーソルを重ねることで，対応するコードを参照できる．また，逆も可能である．このようにブロックとテキストコードの双方向の参照を可能にすることで，ホスト言語の理解を深めテキストベースのプログラミングへの移行を支援する．

3.3. ブロックを用いたプログラミング

視覚的 DSL ではブロックベースのプログラミングを実現した．初学者は本 DSL により効率的にプログラミングできる．その理由を以下に述べる．

ブロックはドラッグ&ドロップなどで容易に操作できる．ユーザはブロックの組み合わせでコードを記述し，ブロックを切り離すことでコメントアウトのように命令を一時的にコードから削除できる．このように，キーボード入力に不慣れなユーザでもマウス操作と最小限のパラメータ入力でプログラミングが可能である．

次に，ブロックはそのはたらきと引数の意味を自然言語で示す．ブロックを日本語で記述することで英語が苦手なユーザでも容易に読み書きできる．また，ブロックは高度なアルゴリズムを隠蔽し，抽象的で可読性の高い表現を提供する．例えば，for 文を表す「n 回繰り返す」ブロック（図 2-c）を用いることで，ユーザは初期化式・条件式・変化式の仕組みを学ぶことなく繰り返しを実装できる．

最後に，ブロックの記述には文法やライブラリを学ぶ必要がない．ブロックは，関数や変数などに変換される．そのコードには文法エラーがなく，文法的に正しい方法でのみ組み合わせることができるため，スペルミス，記号の書き忘れ，型の不一致などのエラーが発生しない．

本 DSL では四つの形状のブロックを実装した（図 2）．ブロックの形状はコネクタの有無や位置で分類できる．スタートブロックは下にのみコネクタを持ち，関数定義に変換される．コマンドブロックは上下にコネクタを持ち，関数呼び出しや代入などの式を表す．ルールブロックは上下と内部にコネクタを持ち，for 文などを表す．アウトプットブロックはコネクタを持たず，他のブロックのインプットに埋め込むことで変数や定数，算術式などに変換される．アウトプットブロックとインプットにはいくつかの形状があり，図 2-d は数値型，図 2-e は真偽値型を表す．ブロックとインプットの形状が一致する場合にのみ埋め込むことができるため，初学者も直感的に型を区別できる．インプットへの入力は，アウトプットブロックの埋め込みだけでなく，インプットへ直接値を記述する，またはカラーピッカーなどの GUI を利用できる．GUI によりキー入力を更に減らし，型や値域のエラーのないコードを記述する．

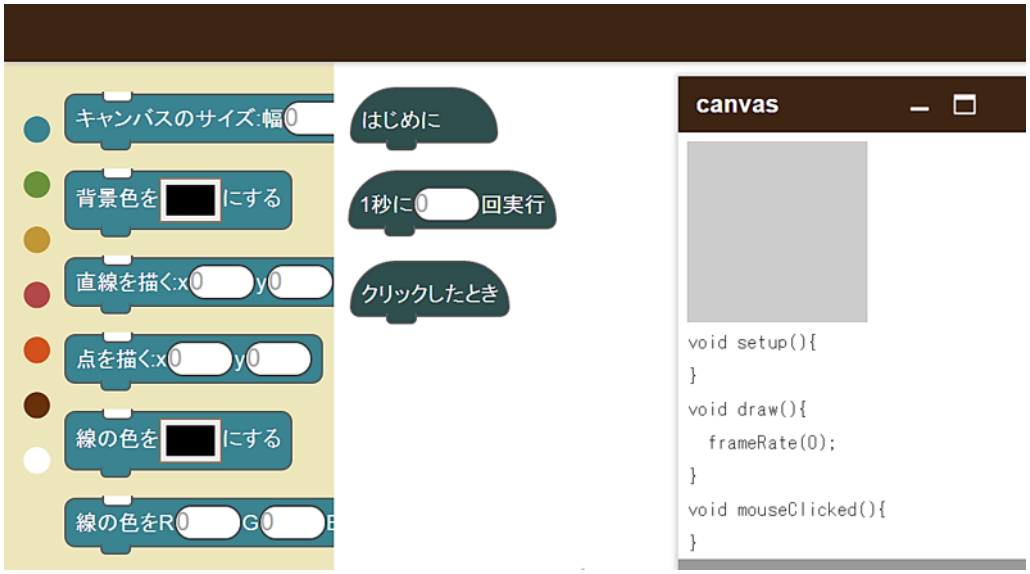

図 1. 左から，パレット，ワークスペース，ポップアップ（キャンバスとプログラム）.

3.4. プログラムのテンプレート

視覚的 DSL ではプログラムのテンプレートを定義できる．テンプレートを用いてプログラムの構造を決定することで，その自由度は低下するが，プログラムのはたらきを明確にし，コードのどの部分を埋めるべきか，どの順番で書くべきかを示す．また，目的に合わせてブロックやテンプレートを選択することで生産性を高めることができる．

以下は Processing でアニメーションを作成するテンプレートについて述べる.「はじめに」ブロックは `setup()`,「一秒に (n) 回実行」ブロックは `draw()`,「クリックしたとき」などのブロックはイベントハンドラに変換される（図 1）．`setup()` はアニメーションの開始時に一度だけ実行され，`draw()` は既定値では 1/60 秒ごとに実行される．これらのブロックをあらかじめワークスペースに配置することで，ユーザにコードの構造とはたらきを示す．これらのブロックは以下のコードに変換される．

```
// variable definitions
void setup(){
  // body of setup
}
void draw(){
  frameRate(n);
  // body of draw
}
// macro definitions
// event handlers
```

このテンプレートでは，変数を全てグローバルとしコードの先頭に置く．これによってスコープ外での変数利用のエラーを減らし変数の管理を容易にする．また，変数定義を除くスタートブロックに繋がっていないブロックは無視され，テキストコードに反映されない．不要なコードをスタートブロックのシーケンスから一時的に切り離すことで，コメントアウトと同様の効果を得られる．

4. 視覚的 DSL の実装

4.1. 概要

本 DSL は HTML，JavaScript，CSS で実装され，web アプリケーションとして提供されるため，インストールやダウンロードが不要であり導入が容易である．本 DSL の環境にはブロックとエディタ，トランスレータが含まれる．ホスト言語の実行系が JavaScript のライブラリである場合，DSL に組み込むことでコードの編集と実行，実行結果の確認を一つのウィンドウで行う．JavaScript はクロスプラットフォームであり様々な分野のライブラリが開発されている．本論文では Processing.js をインタプリタとして Processing のための視覚的 DSL を実装し，ライブコーディングを実現した．

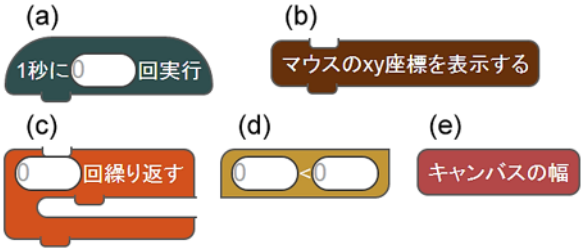

図 2. (a) スタートブロック，(b) コマンドブロック，(c) ルールブロック，(d) 真偽値のアウトプットブロック (e) 数値のアウトプットブロックブロック.

4.2. ブロックの作成

ブロックは，その色と形状，はたらきを示すラベル，変換規則によって定義される．例えば「(n) 回繰り返す」ブロック（図 2-c）を定義する際には以下のコードを記述する．

```
type  : 'rule',
label : '(num) 回繰り返す',
code  :
  'for(int i=0;i<($);i++){ ($tail)}'
```

`type` では start，command などの中からブロックの形状を選択する．`label` はブロックのラベルを指定する．(num) は丸いインプットに置き換えられる．`code` ではブロックの変換規則を定義する．上記のコードは視覚的 DSL の実行前に記述する必要があるため，ユーザはこの方法でブロックを作成できない．そこで，視覚的 DSL の実行中にブロックを作成する機能を開発した．

マクロブロックはユーザによって作成されプロシージャに変換される．ユーザがマクロを作成すると，パレットにマクロブロックが追加され，ワークスペースにマクロ定義ブロックが追加される．マクロブロックは関数呼び出しに変換され，マクロ定義ブロックは関数本体に変換される．マクロのはたらきはマクロ定義ブロックの下にブロックを繋げることで定義されるため，テキストベースの言語の理解を必要としない．マクロブロックを用いることで，類似したコードの繰り返しを防ぎ，コードの可読性を高める．しかし，マクロは既存のブロックの組合せで表現されるため，本 DSL に存在しない機能を追加できない．

そこで，`type`，`label`，`code` を定義することで動的にブロックを作成する機能を実装した．ポップアップで `type` を選択し，`label` と `code` を入力することで，外観を確認しながらブロックを作成する．これによって，DSL に実装されていない機能を持つブロックを追加できる．しかし，この手法にはテキストベースの言語の文法や関数の理解が必要であるため，初学者には適切でない．

4.3. ブロックの変換

本 DSL の開発環境は，ブロックをテキストコードに変換するトランスレータを持つ．トランスレータはブロックの持つ変換規則に従ってこれを様々なテキストベースの言語に変換する．また，これはホスト言語のコードをテンプレートとするテンプレートエンジンとして実装される．3.4 節で述べたプログラム全体の構造を決定するテンプレートだけでなく，ブロックが持つ変換規則をテンプレートとすることもできる．例えば「(n) 回くりかえす」ブロックは `for(int i=0;i<($);i++){($tail)}` というテンプレートを持つ．($) はラベルの (n) にインプットされた値に置き換えられ，($tail) はこのブロックの内部のコネクタに繋がっているブロックの変換結果に置き換えられる．

ブロックは形状によって様々なコードに変換される．スタートブロックは関数定義，コマンドブロックは命令や関数，ルールブロックは if 文など，アウトプットブロックは変数・定数・数式などに変換される．また,「Tips」と呼ばれるブロックは一つ以上の式や関数などの意味的なまとまりのコードに変換される．Tips を用いることによりコードのブロック数が減少し可読性と生産性を高めることができる．以下に Processing における tips の例を示す．

- 「マウスの xy 座標を表示」ブロックをデバッグに利用することで，xy 座標の向きや値を実行結果上で確認できる．また，このような使用頻度の高いデバッグ用ブロックをあらかじめ用意しておくことでユーザが自ら実装する必要が無くなる．

```
text("("+mouseX+","+mouseY+")",0,10);
```

- 「n 回繰り返す」ブロックは初期化式・条件式・変化式を埋めた for 文に変換され，繰り返し回数のみをユーザに入力させる．これにより for 文のしくみを知らない初学者も繰り返しを実装できる．
- 「1 秒に (n) 回実行」ブロックは `draw()` に変換されるが，関数名はそのはたらきを明確に示していない．そこでコード内にフレームレートを設定する命令を追加することでより具体的かつ明確に記述した．

```
void draw(){
 frameRate(n);
 // body of draw
}
```

次に，マクロブロックについて述べる．マクロブロックは関数呼び出しに，マクロ定義ブロックはプロシージャに変換される．例えば図 3 における「枝を描く」はマクロブロックであり以下のようなコードに変換される．

```
branch(mouseX,height,height-mouseY,270);
```

マクロ定義ブロックである「新しいブロック」は以下のようなコードに変換される．

```
void branch(float x, float y,
  float s, float a){
  ...
}
```

ブロックとテキストコードは双方向に参照可能である．ブロック（コード）にカーソルを重ねることで対応するコード（ブロック）の背景色が変化する．これを実現するため，トランスレータは変換時にブロックのシリアライズを行い，対応するコードにも同一の番号を与える．

5. 視覚的 DSL を用いた教材

プログラミング初学者やプログラミング経験者が新たな言語に取り組む際，より効率的な学習のために，パズルや課題を与える．プログラミング言語のイントロダクションは技術的な文書が多いが,．ゲーム形式のチュートリアルでプログラムを記述・実行することで体系的に学ぶことができる．また，技術文書を読むことに慣れていないユーザでも，ゲームやパズルならば取り組みやすい．

課題は少ないブロックと初歩的な機能から始まり，段階的に複雑なアルゴリズムや高度な機能を学ぶ．特に，プログラミング初学者は言語機能を学びながらプログラミングのコンセプト（順次実行・繰り返し・分岐）を理解し，汎用的なプログラミングスキルを身に付けることを目標とする．プログラミング経験者が新たな言語に取り組む際には，ホスト言語の機能を学びながらテキストコードを参照し，テキストベースへのプログラミングへと徐々に移行する．

課題を複数のレベルに分割することで学習者のプログラミングスキルに合わせた課題を提供する．また，課題の難易度を抑え短い時間で多くの課題を解かせることで，学習者の達成感を引き出す．課題の分割の例には，様々なブロックを一度に与えるのではなく一つの問題で一つのブロックを紹介する，または複雑なアルゴリズムを単純化し，ステップごとに問題を作成するなどの方法がある．

プログラミングのイントロダクションには，課題だけでなくサンプルコードを提供することもある．サンプルコードは視覚的 DSL の導入後すぐに実行できる形で提供されるため，プログラミングスキルやホスト言語の機能などの知識を必要としない．ユーザはサンプルコードを読み，実行し，結果を観察することで視覚的 DSL の仕組みやホスト言語の機能を学ぶ．また，サンプルコードを編集することでコードの理解を深める．6 節においては，Processing のための課題の例を紹介する．

6. 視覚的 DSL の他言語への適用

本節では視覚的 DSL をテキストベースのプログラミング言語に適用する手法について述べる．テキストベースのプログラミング言語を習得するには，プログラミングのコンセプトを学び，その文法と機能を理解する必要がある．特に，初学者がテキストベースでプログラミングをする際には，スペルミスなどのバグ，開発環境の導入の難しさなどもハードルになる．そこで，テキストベースの言語のフロントエンドとして視覚的 DSL を適用する．これによって，ユーザは文法を理解する必要がなくなるため，アルゴリズムの構築に集中できる．視覚的 DSL は web アプリケーションであるため導入が容易であるとともに，実行系

を組み込むことで自動でコンパイルし，同じウィンドウに実行結果を表示する．これによって，実行のためのユーザのアクションを減らし効率的にプログラミングできる．

視覚的 DSL をテキストベースの言語に適用する際には，そのブロックとテンプレートを定義し課題を作成する．以下，視覚的 DSL の Processing への具体的な適用の手法について述べる．Processing は Java をベースに開発されたメディアアートのためのプログラミング言語であり，簡潔な表現で視覚的な結果を得ることができる．Processing のための視覚的 DSL によって直感的な操作でエラーのないコードを記述でき，またそのブロックは日本語で表現されるため英語が苦手なユーザでも容易に読み書きできる．プログラミング経験者が Processing を学ぶ際には，Processing の文法を学ぶことなくその機能のみを理解し使用できる．また，ブロックから変換されたテキストコードを参照することでテキストベースの記述への移行を支援する．

図 3 にサンプルプログラムを示す．以下，Processing のためのブロックとテンプレートについて述べる．視覚的 DSL で表現する Processing のコードは，図 2 の四種類のブロックからなる．関数定義はスタートブロック，命令や代入はコマンドブロック，構造文はルールブロック，変数・算術式はアウトプットブロックで表現できる．上記のブロックで表現できない構文がある場合新たなブロックの形状を定義する．例えば，break 文は下にコネクタのないコマンドブロックで表現することができる．また，tips として draw() に変換される「1 秒間に (n) 回実行」ブロックを定義した．これは，学習者が draw() のはたらきを理解しにくく，またその関数名がそのはたらきを明確に示していないからである．Tips を多数定義することでコードが抽象化しコード中のブロック数は減少するが，その種類が増えることでパレットの閲覧性が下がる．

3.4 節では 2D アニメーションのための Processing のテンプレートを示した．Processing でアニメーションを実装するには setup，draw の二つの関数が必要であり，変数定義は変数の利用よりも前に記述する必要がある．また，作成するアプリケーションの種類によってテンプレートを使い分けブロックを選択的に提供することでバグのないコードを記述することができる．例えば，2D と 3D のレンダリングモードの切り替えのコードをテンプレートに記述し，xyz の値を引数に持つブロックのみを提供することで，異なるモードの命令を記述するミスを予防する．

最後に，視覚的 DSL による Processing の課題について述べる．課題の目的は，プログラミングのコンセプトを学ぶこと，および Processing の機能を学ぶことである．例えば，以下の課題を学習者に提供する．

正しい順番に並べ替えよう:
「黒い円を描く」「灰色の円を描く」「白い円を描く」というブロックを与え，これを正解の図と同じ結果が得られるように並び替える．これによって，一つ目のプログラミングのコンセプトである順次実行を学ぶ．

マウスで線を引こう:
「はじめに」「一秒に (n) 回実行」ブロックを与えることでアニメーションを作成し，および「マウスの X 座標」「マウスの Y 座標」ブロックを与えることでカーソルの位置を読み取る．ユーザはマウスの軌跡を描くことで，Processing におけるアニメーションの作成方法を学ぶ．

7. 考察

視覚的 DSL は，言語の習得と開発環境の実装のコストを削減する．一般に DSL はユーザの少なさから開発環境やリファレンス，初学者向けの機能などが整備されにくい．そこで，視覚的 DSL を可読性の低い言語や開発環境が実装されていない言語のフロントエンドとして利用する．視覚的 DSL はホスト言語の文法を隠蔽し，リファレンスとして利用できる．従って，ユーザは文法や機能を事前に学ぶことなくコードを記述し実行できる．また，視覚的 DSL の実行環境にホスト言語の実行系を組み込むことでライブコーディングをサポートする．視覚的 DSL はブロックの変換規則とプログラムのテンプレートを定義することで他の言語へのフロントエンドとして適用できる．したがってこの環境は視覚的 DSL の拡張・制限を行う言語ワークベンチである．視覚的 DSL は他のブロックベース言語と一部同一の機能を持つが，これは様々な言語に容易に適用可能なフロントエンドとしての利用を目的とする．

今後の課題には，大規模なコードの表現とインタフェースの改良，テキストとブロックの双方向モデルの実装，プログラミング教育に向けた更なる機能の実装などが挙げられる．Java などの汎用言語のプログラミングを視覚的 DSL で実現する際には，ブロックの種類の多さと，プログラム中のブロック数の多さが問題となる．本研究ではパイメニューを用いてブロックの探索を改善し，tips やマクロを用いてプログラム中のブロック数を削減した．ブロックの探索では，パイメニューにおけるブロックの分類の改善や，ブロックの検索機能が必要とされる．また，ワークスペースのサムネイルやナビゲーションなどを用いて，プログラム全体を把握する必要がある．

本 DSL はコードをブロックと同様の DOM 構造で表現することで双方向の参照を可能にしたが，テキストコードからブロックの変換はサポートしていない．テキストとブロックの双方向モデルを実装することで既存のテキストコードをブロックで表現し課題の作成を支援する．

プログラミング教育に向けた機能に，タッチパネルへの対応，教材の作成支援などが挙げられる．プログラミング教育ではタッチパネルデバイスの利用が増加している．視覚的 DSL においても指でブロックをドラッグ&ドロップすることで，より容易に操作可能である．また，学生に課題を解いてもらう実験を通じて，課題の難易度の調整やインタフェースの改良などを行う．

8. おわりに

本論文では，テキストベースのプログラミング言語に変換可能なブロックを用いた視覚的 DSL とその開発環境を実装した．ユーザは自然言語で書かれたブロックを視覚的に組み合わせることによってプログラミングを行う．これは操作性と可読性が高いことからプログラミング初学者でも記述できる．ブロックは文法エラーの無いコードに変換され，そのはたらきを平易な自然言語で示すため，文法やライブラリを学ぶ必要がない．

ブロックは自然言語で書かれたラベル，形状と変換規則によって定義される．Tips はスニペットに変換され，ユーザによって定義されたマクロブロックはプロシージャに変換される．Tips とマクロによってコードのブロック数は減少し，より抽象的かつ意味的な表現が可能になる．目的に合わせたブロックを追加・削除することで，DSL を拡張・制限し生産性を高めることができる．

本 DSL はテキストベースの言語の汎用的なフロントエンドである．本論文では視覚的 DSL を Processing に適用し，そのテンプレートとブロック，開発環境，教材を作成した．Processing のための視覚的 DSL のターゲットユーザはプログラミング初学者および Processing 初心者である．この教材の目的はプログラミングのコンセプトを理解し習得すること，および Processing の機能を理解することであ

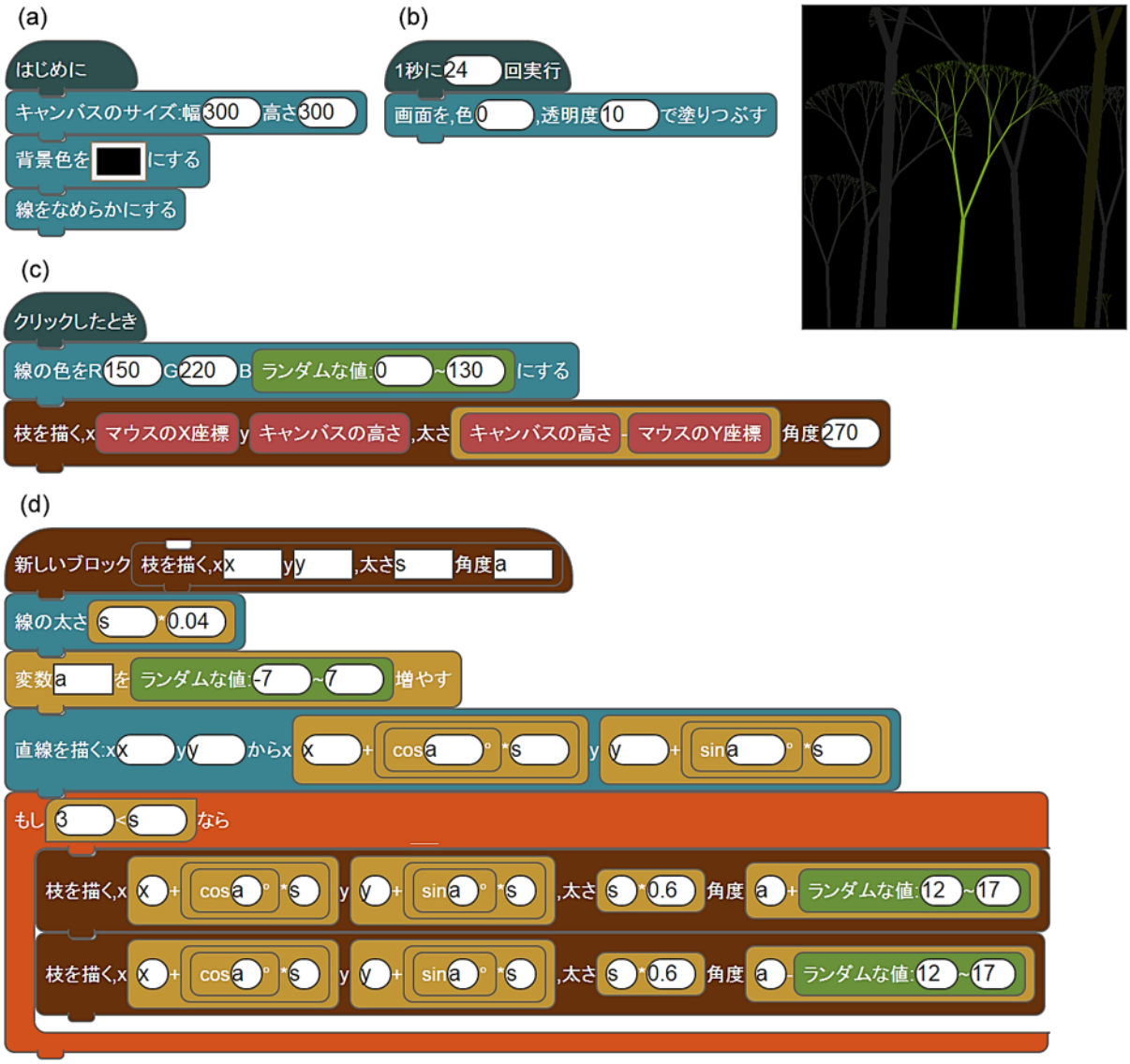

図 3. (a)「はじめに」ブロックは setup() を，(b)「一秒に 24 回実行」ブロックは draw() を，(c)「クリックしたとき」ブロックはイベントハンドラ，(d)「新しいブロック」は branch() の定義を表す．右上は実行結果．

る．教材は学習者のプログラミングスキルに合わせて，基本的な機能から高度な機能に段階的に発展する．

今後の課題には，大規模なコードの表現，テキストとブロックの双方向モデルの実装，プログラミング教育向けの更なる機能の実装などが挙げられる．

謝辞

本研究を遂行し修士論文をまとめるにあたりご指導ご鞭撻いただきました細部博史教授，佐々木晃准教授，東京工業大学・脇田建准教授に感謝の意を表します．

参考文献

[1] L. Allison, *A practical introduction to denotational semantics*. Cambridge University Press, 1987.

[2] B. Victor. (2012) Learnable programming: Designing a programming system for understanding programs. [Online]. Available: http://worrydream.com/LearnableProgramming/

[3] C. Reas and B. Fry. (2001) Processing.org. [Online]. Available: https://www.processing.org/

[4] B. Harvey, *Computer Science Logo Style Volume 1: Symbolic Computing*. The MIT Press, 1997.

[5] M. Resnick *et al.*, “Scratch: Programming for all,” *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.

[6] N. Fraser *et al.* (2012) blockly - a visual programming editor. [Online]. Available: https://code.google.com/p/blockly/

[7] D. Elza. Waterbear. [Online]. Available: http://waterbearlang.com/

[8] N. J. Homer, M., “Combining tiled and textual views of code.” *IEEE VISSOFT*, pp. 1–10, 2014.

[9] C. Done. Try Haskell. [Online]. Available: http://tryhaskell.org/