ビスケットの教育向けコンパイラ

原田 康徳[1,a)]

概要：プログラミング入門ツールビスケットを体験した子どものうちで，もっと先のレベル（文字の言語）に進みたくなる子に対して，どのようにしてビスケットから文字の言語へ移行させるかが一つの課題であった．そこで，ビスケットのプログラを Javascript に変換する教育向けコンパイラを提案する．生成されるコードはプログラミング入門を意識したもので，条件判定，配列，オブジェクトなどの概念が段階的に使われる．たとえば，絵が 1 つしか動かさないプログラムの場合は，シンプルに絵の座標を大域変数で表現し，絵の数が増えてくると配列を使う．カリキュラムに従って，ビスケットのプログラムを段階的に複雑にしてゆくことで，段階的に学ぶのに Javascript のコードが生成されるのである．

キーワード：プログラミング教育, ビジュアルプログラミング, コンパイラ

1. はじめに

ビスケット [1], [2], [3] は，絵を描いて並べるだけでプログラムが作れるビジュアルプログラミング言語である．2003 年に開発され，その後様々な改良が行われてきた．ビスケットの狙いは，プログラミングスキルそのものの習得よりは，コンピュータの可能性と楽しさをプログラミングを通じて体験してもらうことにある．指導法は，落ちこぼれが出ないような工夫がなされ，現在では毎年 3000 人近くの子どもたちに教えるまでになっている．

一方，ビスケットのデザインは，コンピュータやプログラミングを極限にまで単純化してしまっているとも言える．そのため，ビスケットを習得した子どもがその後どのようにして「本物の」言語を習得できるようになるのかが課題ではあった．もちろん，すべての子どもが「本物の」言語を習得すべきという点は議論が分かれることであろうが，少なくとも将来専門の道に進みたいという子どもには，その道を閉ざすようなことがあってはならない．

ここでの課題は，ビスケットを何時間か経験しビスケットの動作について理解している人に対して，どのようにして文字の言語を教えるか，ということである．

2. 教育向けコンパイラ

ビスケットの教育向けコンパイラを開発した．これは，ビスケットのプログラムと同じ動作をする Javascript のプログラムを生成する．一般のコンパイラと異なり，プログラムの部分だけでなくそれを動かすデータまでも含めて変換する．データ（最初の絵の配置）を変更しただけで，生成されるプログラムは異なってくる．

このコンパイラが教育的と言っている理由は，生成されるコードが効率以外の目的で設計されているからである．たとえば，Javascript の配列を使う場合と使わない場合のコードが生成できる．

[1] NTT
[a)] viscuit@gmail.com

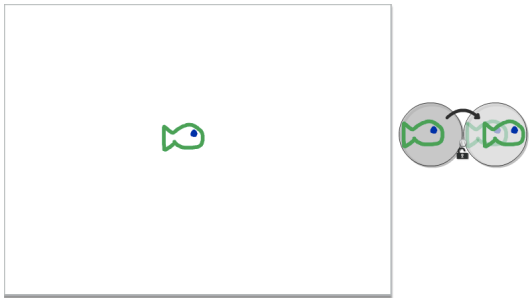

図 1 一つの絵が正確に右に動く

使わない場合は，大量の大域変数と重複したコードが現れるが，そのひどさを見せることで，配列が重要であるということを学ばせる．

あわせて，このコンパイラを用いたカリキュラムが重要である．そのため，ここでは想定したカリキュラムを示しながら，コンパイラがどのようなものであるかを述べて行く．最後に，それに必要なコンパイラの仕様と実装を行う．

2.1 動きの基本

まず，絵を一つかき，それを 1 つだけステージに入れて正確に右に移動する，というプログラムをつくる（図 1)．

それがコンパイルされると，

```
var p, x, y;
function init() {
  p = new Picture('134,274 117,269 ..略..');
  x = 300;
  y = 250;
}

function step() {
  clear();
  x = x + 5;
  draw(p, x, y);
}
```

という形のコードに変換される．最初に init が一度だけ呼び出され，その後 step が 200ms ごとに呼び出される．Picture の部分にはここで描いた絵のテキスト表記が入る．clear や draw は外部で定義された関数で，clear が画面を消去，draw は x,y の位置に図形 p を表示するものである．毎ステップごとに大域変数 x の値が 5 ずつ増加するので，絵が右に動くアニメーションになる．

図 2 一つの絵が正確に下に動く

図 3 一つの絵が斜めに動く

その後，ビスケットのプログラムを修正し，絵が右に動く速度を変化させると，生成されるコードでは 5 の数値が変化する．絵を左に動かす場合は減算になる．また，テキストエディタを用いて数値を直接変更すると，直感的な速さではなく，きちっと決めた速度で動かすこともできる．

次に正確に下に動かすプログラムを試してみる (図 2)．

これは

```
function step() {
  clear();
  y = y + 7;
  draw(p, x, y);
}
```

のように，今度は変数 y が増加するプログラムとなっている．上に動く場合は減算することも確認する．

次に，ビスケットで絵を斜めに動かすのを作り試してみる (図 3) と，

```
...
  x = x + 5;
  y = y - 4;
...
```

のような形になる．様々な方向や速度で動かすことで，この 2 つの数値 5, -4 が変化するところを確認する．

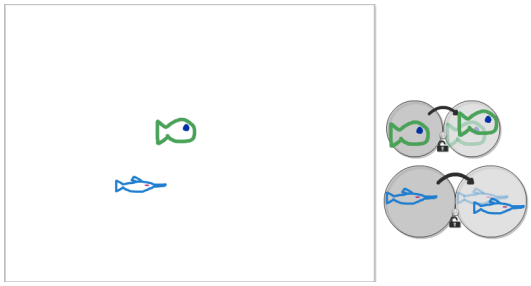

図 4 二つの絵が一つずつ動く

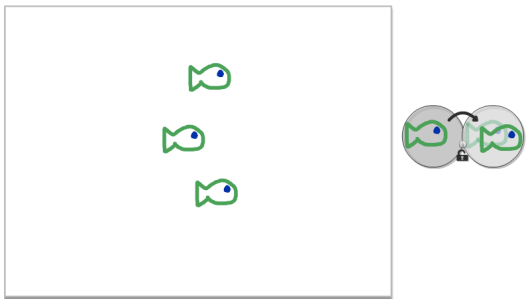

図 5 同じ絵が 3 つ動く

2.2 二種類の絵

次に新しく絵を描いて，二つの異なる絵がステージに一つずつ置かれて，それらが動いているという例を作る (図 4).

```
var x1,y1,p1;
var x2,y2,p2;
function init() {
  p1 = new Picture('...');
  x1 = 120;
  y1 = 53;
  p2 = new Picture('...');
  x2 = 80;
  y2 = 95;
}
function init() {
  clear();
  x1 = x1 + 5;
  y1 = y1 - 4;
  x2 = x2 + 2;
  y2 = y2 + 12;
  draw(p1,x1,y1);
  draw(p2,x2,y2);
}
```

のようなコードになる．動かす絵が増えると，その絵の位置，方向，形を保持するための変数が増える．最初に絵を置く場所を変えると，init の中の数字がかわる．ビスケットのインタフェースでは絵を縦にそろえて置くことはできないが，コード上の数字を直接変えることで簡単にできる．また，二つの絵の速度が正確に 2 倍になるようにするなど，数値を直接指定した方がよい場合があることも試してみる．

2.3 条件判定

ここまでは，絵が画面の端に行った場合，そのまま絵が消えて行くモードで実行していた．ここで，絵が端に行った場合は，画面の反対側から出てくる，というモードに切り替える．簡単のために画面には絵が一つだけで，正確に右に動いているとすると，

```
x = x + 5;
if (x > 512) {
  x = x - 512;
}
```

というコードが生成される．絵が右に動く（x が増える）ことがわかっているため，画面の右端かどうかをチェックする条件判定があればよい．右端に到達したら画面幅分だけ左に飛ぶ．これで条件判定がどういうものかがわかる．

同じように二つの絵にしてみると，条件判定が二つ必要になる．

2.4 配列

配列を教える前に，配列を使わないコードを見せる．まずは，同じ絵を 3 つ動かしてみる (図 5).

```
(変数宣言は省略)
function init() {
  p = new Picture(...);
  x1 = 120;
  y1 = 53;
  x2 = 80;
  y2 = 95;
  x3 = 150;
  y3 = 78;
```

```
}
function step() {
  clear();
  x1 = x1 + 5;
  y1 = y1 + 4;
  x2 = x2 + 5;
  y2 = y2 + 4;
  x3 = x3 + 5;
  y3 = y3 + 4;
  draw(p,x1,y1);
  draw(p,x2,y2);
  draw(p,x3,y3);
}
```

3 つの絵の最初に置いた場所がそれぞれ異なるので，init の中のコードが増えるのは仕方が無いが，step の中のコードは 5 を足す，4 を足す，と同じ計算を違う変数に適用しているだけで冗長である．この先，絵の数をもっと増やすとどういうことになるか想像させる．

ここで，コンパイラに配列を使用するというオプションを使いコンパイルする．すると次のようなコードになる．

```
function init() {
  p = new Picture(...);
  x = [];
  y = [];
  x.push(120);
  y.push(53);
  x.push(80);
  y.push(95);
  x.push(150);
  y.push(78);
}
function step() {
  clear();
  for (var i=0;i<x.length;i++) {
    x[i] = x[i] + 5;
    y[i] = y[i] + 4;
    draw(p,x[i],y[i]);
  }
}
```

こうすることで，動かす絵を追加しても，初期値の部分が増えるだけで，繰り返しのコードは増えない．画面がループするようにすると，2.3 節のような if 文が入るが，この場合も同様である．

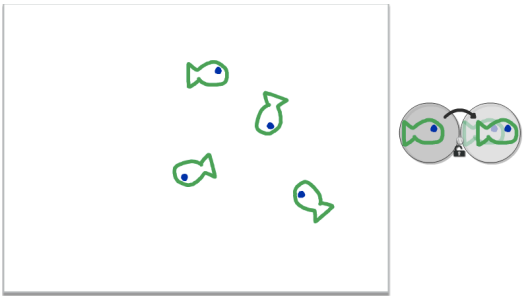

図 6 様々な方向に絵が動く

また，2.2 節のように動かす絵の種類を増やした場合は，配列名を x1, x2 のように種類だけ増やして対応する．この時点では種類が増えた場合の冗長性には対応できていない．

2.5 相対座標

この部分は，学校で習っていないことを使うので，対象の学年によってオプション的に教える内容である．図 6 のように画面上に色々な向きで絵を置いた場合，プログラムで絵が右に進むようになっていても，実際には絵はその絵にとっての相対的な方向での右に動くようになる．このため，変数として絵の向いている方向をベクトルで示す方法をとる．このコードは次のようになる．

```
function init() {
  p = new Picture(..);
  ...
  x.push(120);
  y.push(53);
  a.push(-0.02);
  b.push(0.1988);
  ...
}
function step() {
  clear();
  for (var i=0;i<x.length;i++) {
    x[i] = x[i] + 5*a[i];
    y[i] = y[i] + 5*b[i];
    if (x[i] < 0)   x[i] = x[i] + 512;
```

```
    if (x[i] > 512) x[i] = x[i] - 512;
    if (y[i] < 0)   y[i] = y[i] + 384;
    if (y[i] > 384) y[i] = y[i] - 384;
  }
}
```

ここで，新たに a と b という配列が導入されたが，これは絵の回転と縮尺を表すベクトルである．ここでは右に速度 5 で動くプログラムなのでこうなるが，斜めに動く場合は

```
x[i] = x[i] + 5*a[i] - 4*b[i];
y[i] = y[i] + 5*b[i] + 4*a[i];
```

のようになる．画面がループする場合，端まで進んだときに反対側に戻る処理は 4 つのすべての境界で生じるため，4 つの条件式が必要である．

さらに，絵が直線運動だけでなく，回転移動を伴う場合は，ベクトルの方向も更新するため，たとえば，

```
var aa = 0.6774*a[i] - 0.7356*b[i];
var bb = 0.7356*a[i] + 0.6774*b[i];
a[i] = aa;
b[i] = bb;
```

のような行列のかけ算が行われる．

これらのコードはプログラミング言語習得には本質できはない上に，線形代数の部分が多少必要になるため，教えなくても良い．

2.6 衝突判定

ビスケットのプログラムの特徴は，絵同士が衝突判定で特別な動きをさせる点にある．たとえば図 7 では，ロケットがまっすぐ飛ぶプログラムと，星に衝突したら向きを変えるというプログラムが入っている．これをコンパイルすると次のようなコードが生成される．

```
function init() {
  p1 = new Picture(...);
  p2 = new Picture(...);
  x1 = []; y1 = [];
  x1.push(353); y1.push(108);
  x1.push(202); y1.push(57);
  x1.push(226); y1.push(326);
  x2 = 357; y2 = 326; a2 = 1; b2 = 0;
}
```

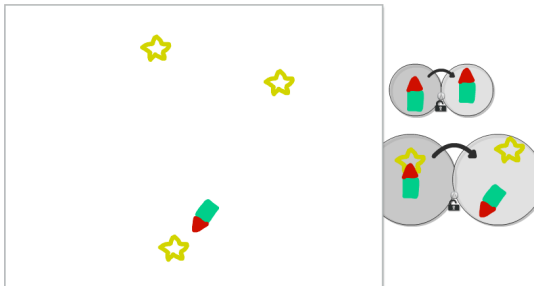

図 7 衝突判定

```
function step() {
  clear();
  var xx = x2 - 2.25*a2 + 133.25*b2;
  var yy = y2 - 2.25*b2 + 133.25*a2;
  var mind = 40;
  var mini = -1;
  for (var i=0;i<x1.length;i++) {
    var d = distance(xx, yy, x1[i], y1[i]);
    if (d < mind) { mini = i; mind = d; }
  }
  if (mini>=0) {
    ... // 星とロケットがぶつかったとき
  } else {
    ... // ロケット単独で進むとき
  }
  ...
}
```

このプログラムは x1,y1 が星の座標，x2,y2 がロケットの座標を示してる．ロケットは 1 台しかないため配列は使われていない．このプログラムで重要なのは，ロケットが星とぶつかる判定をする部分である．step の xx,yy は今のロケットから相対的に見たときにぶつかるであろう星の中心座標である．その座標から距離が一番小さくかつ 40 より小さい星が存在するかを for ループで探している．見つかった場合は mini が星のインデックスであるから，ここで，その星とロケットの新しい座標を計算する．見つからなければ，ロケットが単独で進むプログラムができる．

2.7 プログラムの重複部の関数化

プログラムで絵の種類・数を増やして複雑にしてゆくと，プログラムの中に似た記述が見つかる．同じ記述と異なる記述を分離して，一つの関数として定義し置き換える．これまで関数は，組み込みの機能として登場しているけれども，重複する記述を省略する目的での関数はでてきていない．関数の中身について意識されるのはこれが初めてである．

2.8 絵の型の導入

これまでは，絵の種類ごとに変数を変えていた(星は x1,y1 ロケットは x2, y2 など)．絵の型を保持する変数を新たに導入して，種類ごとに異なっていた変数を統一することが可能となる．たとえばこのようなものである．

```
x[i] = x[i] + a[i]*dx[p[i]] - b[i]*dy[p[i]]
```

絵の種類を p の配列でしめし，絵の種類ごとの移動ベクトルを dx, dy で表現している．この導入により，絵の種類が増えても p, dx, dy を増やせば良いだけになる．

2.9 オブジェクト指向

絵の数が増えたり減ったりするプログラムまで拡張すると，バラバラの配列として管理するのは不便になる．そこで，オブジェクトを導入して，オブジェクトが入った 1 つの配列で管理する．

```
o[i].x = o[i].x
  + o[i].a * p[o[i].p].dx
  - o[i].b * p[o[i].p].dy;
```

のようになる．ここで o はステージに置かれたオブジェクトの配列，p は絵の定義の配列である．

3. コンパイラの実装

ビスケットのコンパイラは，ビスケットの内部表現である XML から，javascript を生成する．コンパイラが生成するコードの種類に応じて次のようなオプションがある．

- 配列をつかわず表現．すべて単純な大域変数．コードはベタ．
- 同じ種類の絵のみ配列を使用．同じ種類の絵のみ繰り返しのコード．
- 関数を導入して，コードの重複を一部削除．
- 絵の型を導入して，より多くの重複を削除．
- オブジェクトを導入して，完全に重複を削除．

また，以下の仕様はオプションに関わらず常に有効である．

- 部分計算をし，定数 0, 1 のかけ算, 0 の加算，負の数の加算などは，式をシンプルにする．
- 論理上不要な条件の削除（x が増加しかしないのに x ¡ 0 の条件があるなど)．
- 要素が一つしか無いことがわかっている配列を単純な変数にする．

実装の結果 python で 1000 行弱である．教育用という視点からは，このコンパイラ自身も見せるつもりで作成してある．

4. 考察

このコンパイラを用いた実際の教育はまだ実施していない．同じ対象（ビスケットを知っているけれど，文字の言語には進んでいない子ども）向けに，ビスケットのプログラムを見せ，それと同じ動きをする（著者があらかじめ制作した）Javascript のコードを見せ，それを子どもが改造することで動きを確認するという講座を実施したことがある．ビスケットの動作を知っているからできる指導法であり，有効であると感じた．この経験をヒントに，今回のコンパイラの制作に思い至った．

このツールはカリキュラムと一体である．カリキュラムは，困らせてから（必要性を感じてから）それを解決する方法や機能を提示する，という方針をとっている．つまり最初に失敗させ，そこから学ばせる．これを写経的な方法で実践するのはモチベーションを維持する上で非常に難しいと思われるが，機械に生成させることでその欠点を補っている．

一方で，生成されたコードを直接編集して，ビスケットでは難しかった精密なコントロールを可能にする点も特徴である．これがあることでビスケットで大半のことができるのに，わざわざ文字の言語をつかうモチベーションもつくりだせる．

python で作られたコンパイラを見せることも重

要だと考えている．たった，1000 行くらいであるし．複数のプログラミング言語が絡み合う様子をプログラミングの入門の段階で知ることは重要である．

生成されたコードは，ビスケットの意味を学ぶ上で重要な情報である．絵で定義された漠然としたものが厳密に定義されることの面白さ，大切さを知ることでもある．たとえば衝突判定は，判定の度に他のすべての絵をスキャンして探すという処理が必要なのであるが，普通はビスケットの 1 ステップでそんな複雑なことをしているとは誰も思わないはずである．ここでは 2 つの絵の位置関係しかコンパイルしていないが，3 つの絵の位置関係にすると，実際のビスケットでは 2 つの最適な組み合わせを探索することになるので，極めて重い処理をしている．そういうことを知るきっかけにもなる．

カリキュラム後半の衝突判定や型の導入などは，プログラミング言語の教育というよりはプログラミング手法の教育として考えるべきである．プログラミング手法は他にも沢山存在するが，このやりかたを応用すると，まずは何か動いているもので遊んで，その動きを十分理解した上で，その中身のプログラムを見せる，という手順になるかと思う．その手順を繰り返すことで，様々なプログラミングの技術が身に付いて行く．

5. まとめ

ビスケットの課題であった，これを学んだ後にどのようにして文字の言語へつなげるかという問題について，ビスケットのコードから適切なコードを生成する教育用コンパイラを用いた指導法を提案した．

今後はこれを Web サイト上で公開し，高校，大学等の授業でも使えるような環境を整備する予定である．

現時点ではビスケットのサイトの「うごく絵本をつくろう（ベータバージョン）」というリンクから起動するビスケットは，このコンパイラを応用して動いている．ソースコードは一応読めるが，段階を意識したものではなく，逆にうごく絵本用に複数ページの扱いや，タッチイベントの拡張などがされている．

参考文献

[1] Yasunori Harada, Richard Potter : *Fuzzy Rewriting – Soft Program Semantics for Children –*, HCC 2003, IEEE (2003).

[2] 原田康徳: 体験型ワークショップ用ソフトウェアの開発, 第 50 回プログラミングシンポジウム, 情報処理学会 (2009).

[3] 原田康徳，勝沼奈緒実，久野靖: 公立小学校の課外活動における非専門家によるプログラミング教育，情報処理学会論文誌 Vol.55 No.8 (Aug. 2014).