Pietのどうじんし

いっと☆わーくす！

Contents

Chapter 1

Pietの自動生成

1.1　はじめに

こんにちは、NoNameA 774です。KMC(京大マイコンクラブ)というPietサークルに入っているのですが、サークルの人々がPietを手で(書いてる|描いてる)のを見て、絵としての意味を持つ方法以外でPietを人間が描くのは不毛だと思い、Pietの自動生成を行いたいと思った次第です。今回一度ぐらいサークル参加をしてみたいとコミケに初申し込みしてみたら、運良く当選したのでこの本を書いています。

前回のコミケ(C88)でも同じようなペーパーを出したので(https://nna774.net/piet/)、それの~~焼き直し~~続きのような感じです。前回のペーパーは二日目のアース・スター作品のゾーンに割り当てられた友人のサークルにて配布したのですが、完全にアウェーな中でも何枚か配ることに成功しました。もしその際手にとってくださった方がいれば、ありがとうございました。

KMCの部誌の記事としてこの本を出さなかった意味はあるのか、という話もありますが (今回KMCの部誌の記事として遠野へ旅行した旅行記を書きました。今これを読んでる時点ではほぼ手遅れである可能性が高いですがそちらもよろしくお願いします)、一度自分のサークルで参加したかったということでよろしくお願いします。

今回のPietの自動生成のレポジトリですが、https://github.com/nna774/piet-automataになっております。最新情報等はこちらを確認してください。Pull request welcome.

1.2　前提

今回この記事はプログラミング言語Pietについて書いているので、Pietそのものについての解説もすべきな気がするのですが、ここでは概要を説明するに留めます。

Pietについてより詳しくは公式サイト(http://www.dangermouse.net/esoteric/piet.html)を見る、もしくはうちの部員の書いたスライド(http://www.slideshare.net/KMC_JP/piet-46068527)を参照するなどお願いします。特にこのスライドは一度目を通しておいておいてくださると大変話が早いのでよろしくお願いします。「Piet」でGoogle検索するとWikipediaの次ぐらいに出てくる「Pietのエディタを作った話」というやつです。

また、このスライドに書かれているPidetというPietのエディタですが、現在OSSになっていてhttps://github.com/kndama/Pidetから入手可能となっています。Pietを触ってみようという方はこれを使うか、とりあえず試してみるだけならPietDev(http://www.rapapaing.com/blog/?page_id=6)を利用するのが良いと思います。

あとこれはPietに関係はありませんが、上記のスライドに出てくるUnambiSweeperですが、現在作者によりアンドロイドアプリとしてリリースもされているので(https://play.google.com/store/apps/details?id=jp.nobody.dnek.unambisweeper(短縮: http://bit.ly/10oMg7J))、ぜひダウンロードして見てあげてください(宣伝)。

1.2.1 言語仕様の概要

Pietは20色の色でドット絵を描くことによりソースを書きます。赤、黄、緑、シアン、青、マゼンタの6色と、それに明、普通、暗の三種類を組み合わせた合計18色に白と黒を合わせた合計20色です。この色と明度はこの順序で後述する差を考えるときには扱われます。

この20色で描かれた図形の上をプログラムポインタ(最初は左上にある)が、Direction-Pointer(以下DP)、CodelChooser(以下CC)と呼ばれる2つのステートを参照しながら移動していくことにより命令を実行します。このプログラムポインタの動き方を説明するのはすこしややこしいので詳細はスライドを参照して欲しいのですが(23-29枚目のところです)、簡単に言うとDPが上下左右の方向を指し、CCがDPの指す方向を向いた時の右左を指定します。プログラムポインタはDPの方向を向き、今いるマスと同色のマスの中で一番その方向に遠いマスを探します。複数ある場合はその中でCCで指定された方向について一番端のものを探し、その一つ前のマスに移動します。

この移動の時に、移動元と移動先の色の“差”と、移動元の面積により命令が決定されます。6色の差と3明度の差(それぞれ循環して考えます。赤から緑の差は2段階で、緑から赤の差は4段階。明から暗は2段階、暗から明は1段階)により、17個の命令が定まっています。

それぞれの命令の割当は省きますが、Pietはこれらの命令によりスタックを操作するスタック指向プログラミング言語となっております。おおまかに分類するとスタック操作、四則演算、否定、比較、CCとDPの操作、入出力の命令があります。

黒ブロックとプログラムの外は移動できないブロックとなっています(仮想的に外側に黒ブロックがあると考えても良いです)。プログラムポインタの次の移動先が移動できないブロックとなっていた場合、DP、CCを変更して新たな移動先を探し、DPとCCの4通りx2通りの8通りを試しても移動できなかった際にプログラムが終了します。

白ブロックにプログラムポインタが移動しようとした時はDPの指している方向にそのまま進んでいき、色ブロックに到達した場合はそこに移動します。この時この移動に割り当てられた命令は実行せずプログラムポインタの変更だけをします。移動できないブロックに到達した場合は、移動しようとしていた白ブロックも移動できないブロック扱いとなり、DPとCCを変更しての探索と移ります。この移動ですが、私は最初「白ブロックに出た時そのまま滑って行き次のブロックに入る」と理解していましたが、それは実は誤りで、実際は色のあるブロックに移動するときはそれで良いのですが、滑っていった結果移動不能ブロックにぶつかった時は正しくなく、滑ったということ自体が無かったことになるのでこうではないです。

この白ブロックを挟んで色ブロックへ移動した場合に命令を実行しないという性質が良い性質で、コード生成はこの性質に大きく頼っています。

1.3 先行

勿論いままでにもPietの自動生成をしている人がいます。

1.3.1 http://www.matthias-ernst.eu/pietbrainfuck.html

基本的にはBrainf*ckインタプリタ。Brainf*ckからPietへの変換もできるようなもの。ソースコードが公開されているわけではないのでよくわからないのですが、まぁ変換の動作としては、Brainf*cuインタプリタがあって、そこにエンコードしたBrainf*ckのソースコードをくっつける用な感じです。

サークルのSlackのPietチャンネルにこのページのURIを投げたところ、高速にインタプリタの部分の小型化が行われました。頭がおかしいと思います(褒めています)。

1.3.2 http://www.toothycat.net/wiki/wiki.pl?MoonShadow/Piet

C-likeなコードからアセンブリlike なコードに変換でき、そのアセンブリlikeなコードからPietへと変換ができる。ソースコードは読めるがPerlで書かれていたのであまりちゃんと読んでいない……。

生成されたコードを見ている限り、メインとなるコードと、そこからジャンプするサブルーチンの列を作るような感じになっているらしいです。今回私が生成するコードと大体似ていますね(似ているといえば似てる ぐらいですが)。

これの良い所は、C-likeなコードからアセンブリlikeなコードを生成できるところで、現状の私のコードはPietの擬似命令からPietへの変換を提供しているだけなので、もう少し高級な言語から擬似命令への変換を行えるようなものを作りたいと考えていますが現状特に実装はありません。

1.3.3 https://jefworks.github.io/mondrian-generator/

これは今回先行のものを探していた時に見つけたのですが、Pietの自動生成 です。あんまり関係ないのですが、なかなか良かったのでどこかに書きたかったけど、特に書くとこが見つからなかったのでここに書いときます。Pietの元ネタとなったピエト・モンドリアンの絵のようなものを生成してくれるページです。

1.4 前回とのDiff

前回のペーパー(https://nna774.net/piet/c88paper.pdf)の時点からの簡単なdiffを書いておきます。詳しくは現在の状態を後述するつもりなのでそちらをご覧ください。

一つ目の大きな変更として前回のペーパーの時点では7x7のサイズのタイルを並べてコードを生成していましたが、3x3のサイズのタイルで並べるようにすることに成功しました。これによって面積で9/49倍程度の最適化に成功しました (小さくする際に幾つかの命令が複数マスに展開されるようになったので厳密にそうではないですが)。

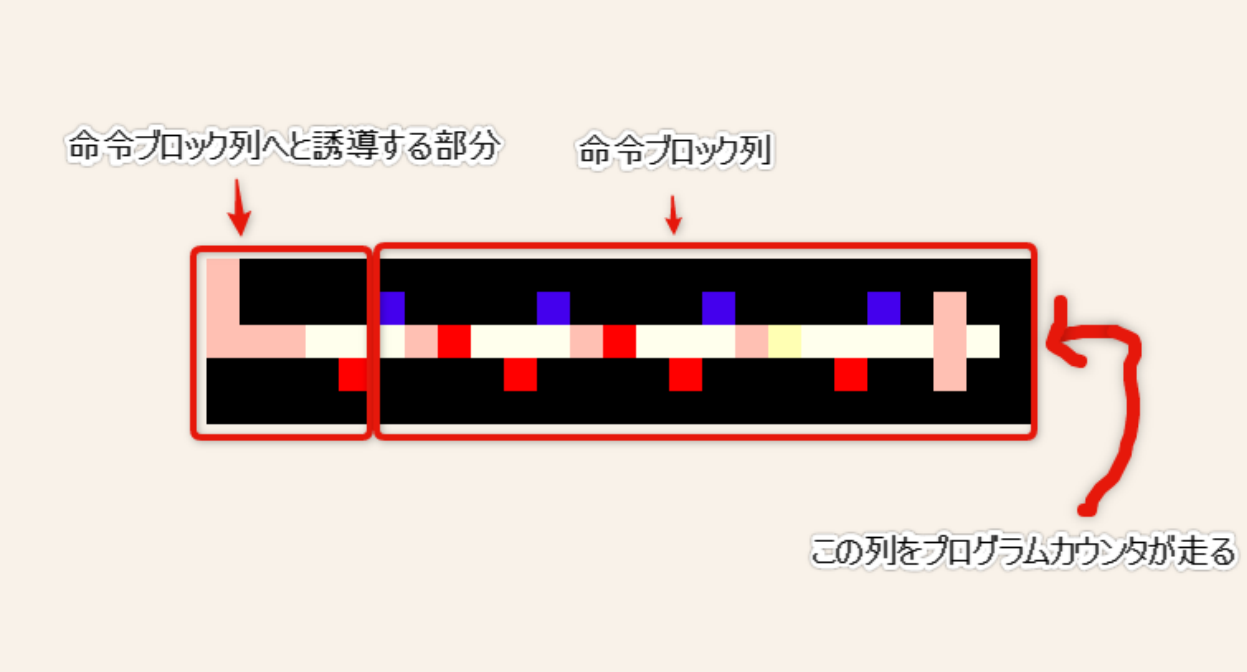
もう一つの大きな変更は、以前まではGoto系の命令の命令数についてO(n)で生成される画像の縦幅が伸びていましたが、上下方向の圧縮を行うコードを入れることによって、max(ジャンプの間にまたぐgoto命令)程度まで縦方向の太さを圧縮できるようになりました。厳密に可能な中で一番小さい縦幅とはなっていないように思いますが、多くの場合で小さくなったので一応良しとしています。

1.5 Pietの生成

Pietを手で描いてみましたか？ Pietを描いたことありますか？ ありますよね。よかったです。どういうふうに描きましたか？ 最初は一直線な絵を描きましたか？ 頭のなかでPietで使える命令を並べて擬似コードを作りましたよね？ というわけで、擬似命令の列からそのままPietへと落とせるようなものを作ろうと思ったのが始まりでした。

Pietのプログラムカウンタの移動の際のルールとして、色マスから色マスへと直接移動した場合は命令を実行するが、白マスを挟んで移動した場合は実行しないというものがあります。この性質を利用して、Pietの1命令を実行する二色のコーデル(ピクセルの論理単位)を置き、それを飛び越えて余分な命令を実行させないための白マスで挟んだものを一つの構成単位(私はブロックと呼んでいます)としました。最初の左上のブロックで、右へと向かう一直線の命令列ブロックが正しく動くように誘導していく感じです。

https://gyazo.com/783f2ae98363220ed6859c9bd97cb4ff

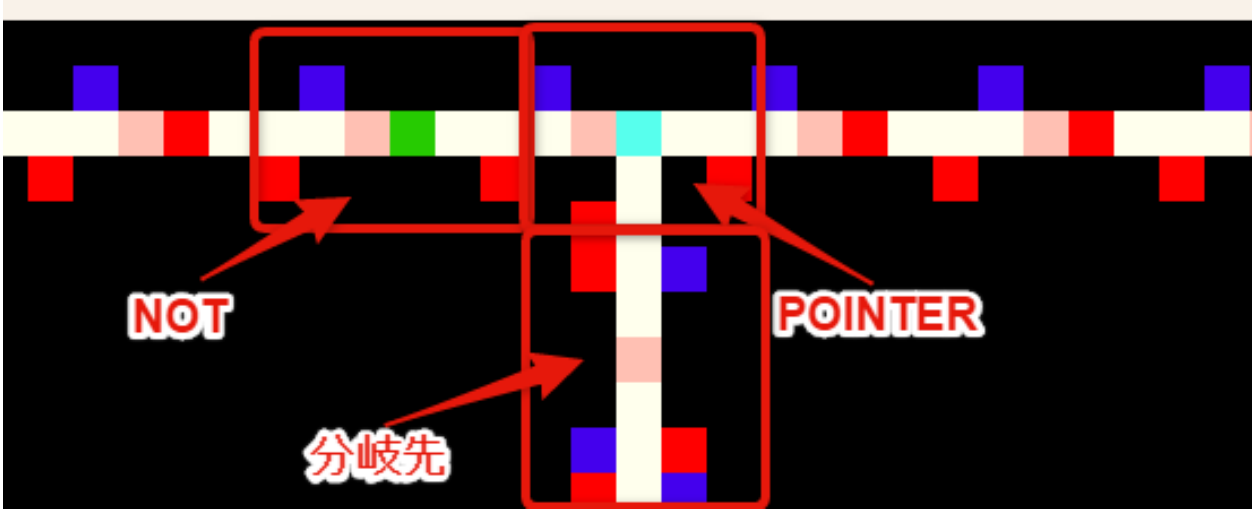


赤と青のポッチは接続方向を表すマーカなので、本質的には不要なので無視してください。

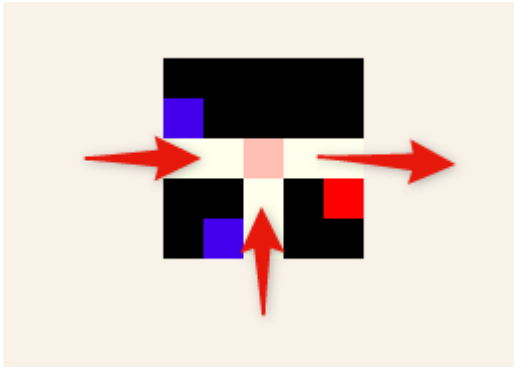
一直線のブロックの列だと、分岐もループもできず、つまり構造化定理の順次しか実現できないので、殆どのプログラムを書くことは出来ません。が、この時点でHello Worldは書けますね。

で、次は分岐を作るわけですが、Pointerという良い命令がPietにはあって、それを使うとDP、つまり進む向きを変えられます。Notという命令もあって、それを使うとスタックの先頭が0の時だけ先頭を1に変え、それ以外の時は先頭を0に変えられます。Pointer命令はスタックのトップの値を見て、その数だけDPを時計回りに回します。これを使えば、スタックの先頭が0の時に分岐するようなものが作れます。

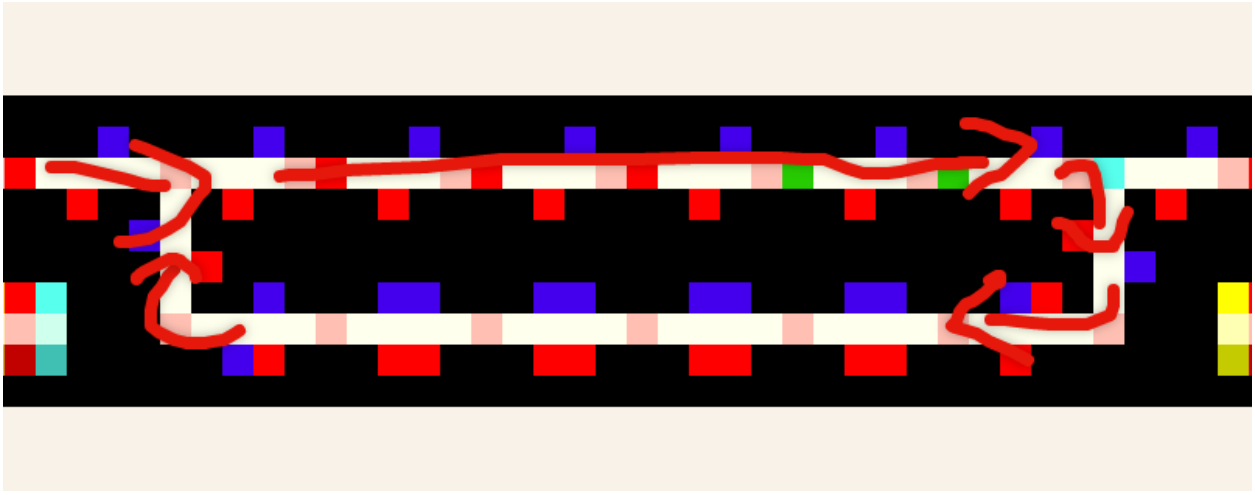
https://gyazo.com/3c7bc0cb0dd41680c6b652ef4486b390



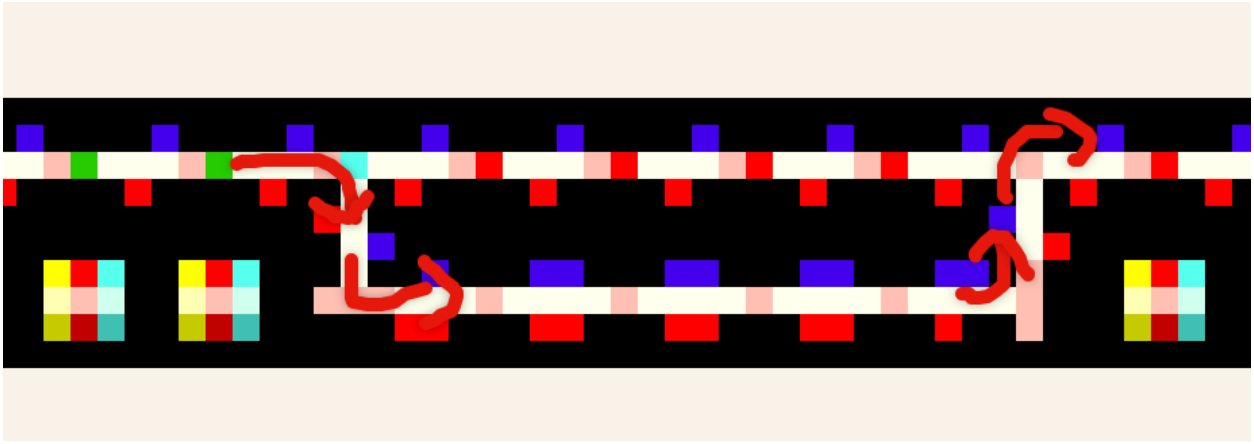
スタックの先頭が0以外の時はそのまま直進し、0の時は下に分岐します。
これで分岐することには成功したので、適当なところに合流したいと思います。
https://gyazo.com/4f893a8b1a52e2de3f9bd70338ebb9b8



このような構造でプログラムカウンタが下から来た場合と左から来た場合に合流してともに右に流すことができるので、ここに飛んで来るジャンプのようなものが書けて、あとはすぐにわかる構造で、以下の様な後ろ向きへの条件付きジャンプができます。
https://gyazo.com/846e9f87a072a65e02e6943e20adc534



これとよく似た感じで、少し考えれば以下のように前向きに条件付きジャンプができます。
https://gyazo.com/83ac7efa68410f18a8cfdfa1ba52d2ee



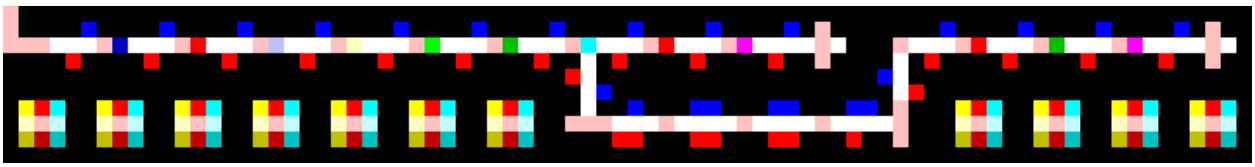
ここまでくれば、後は必要な物を幾つか用意するだけでどんなプログラムだって書くことが可能です。

後はここまでやってきたようなものを使い、javascriptでCanvasの上にベタベタとブロックを重ねてゆきます。ブロックは心をこめて手でPidetを使い打ちます。

以下、生成されるコードの例を幾つか上げます。

数字を1つ入力として取り、それが奇数なら1、偶数なら0を出力するPietです。Piet-automataでは、以下の様な擬似コードを書いて、それをPietへと変換します。命令の詳細はhttps://github.com/nna774/piet-automata/blob/master/README.mdにあるので、そちらを参照してください。https://github.com/nna774/piet-automata/blob/master/tests/even-or-odd.pas

```
INN
PUSH 1
DUP
ADD
MOD
JEZ zero
PUSH 1
OUTN
HALT
LABEL zero
PUSH 1
NOT
OUTN
HALT
```



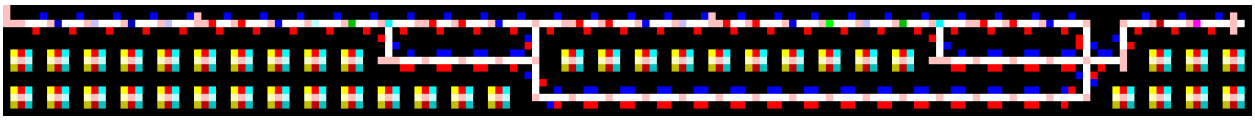
2つ数字を入力として取り、GCDを出すPietです。Cで言うと

```
int gcd(int n, int m) {
  int tmp;
  if (n < m) return gcd(m, n);
```

```
  tmp = n % m;
  if (tmp == 0) return m;
  return gcd(m, tmp);
}
main() {
  int n, m;
  scanf("%d", &n); scanf("%d", &m);
  printf("%d", gcd(n, m));
}
```

のようなことをしています。https://github.com/nna774/piet-automata/blob/master/tests/gcd.pas

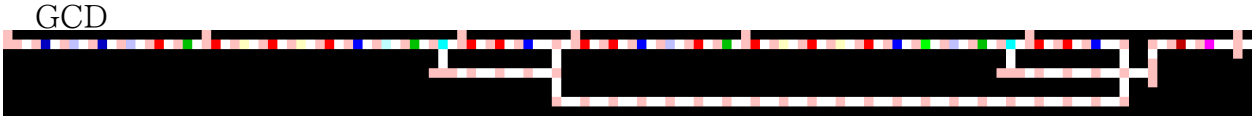
```
INN
DUP
INN
DUP
PUSH 3
PUSH 1
ROLL
GREATER
JEZ S
PUSH 2
PUSH 1
ROLL
LABEL S
# ここでスタックに二つ入っていて、大きいほうが上
PUSH 2
PUSH 1
ROLL
DUP
PUSH 3
PUSH 1
ROLL
# 小さい方は使うのでコピー
MOD
DUP
JEZ END
PUSH 2
PUSH 1
ROLL
JMP S
LABEL END
POP
OUTN
```



実は最新のpiet-automataではブロックのサイズを3x3にすることに成功しているので、以下のようになります(ここまでは説明するときにわかりやすいかと思い5x5のブロックばかり出してきました)。

奇遇判定

GCD



1.6 おわりに

とまぁだいたいこんな感じでPietの擬似命令の列からPietの画像を生成しています。今回はだいぶ実装の詳細は省いたので、詳細が知りたい方はhttps://github.com/nna774/piet-automataをご覧ください。今回書いたことをだいたいそのまま実装して、幾つかの最適化をかけたような感じです。‘/piet’の中にブロックが入っていて、‘config.js’にその対応が入っています。

もう少し高級な言語から擬似命令列を生成できるようなものを作りたいと考えてはいるのですが、まだ実装はありません。

最後に幾つかこのレポジトリに関連したものを友人が作ってくれたので、紹介しておきます。

- https://github.com/primenumber/pasxx 私の擬似命令に拡張命令を追加して、piet-automataで使える列に変換できるものです。サイズが巨大になりがちなので少し厳しいです。
- https://github.com/1995hnagamin/pas-interpreter 擬似命令列をPietに落とさずにそのまま実行できるスクリプトです。

今回初挑戦の拙い文章でしたが、ここまで読んでいただきありがとうございました。

Chapter 2

僕の考えた最強のPiet拡張

2.1　はじめに

murataと申します.TwitterIDは@paradigm_9です. nona7(nona7注: NoNameA774ですがサークルでは主にnona7と名乗っています)さんと同じくKMC(京大マイコンクラブ)というサークルに所属しています. 今回Pietで一つのサークルとして出店するという@nona7さんの挑戦を聞いて,「これは寄稿するしかない!」と勢いだけで寄稿します. 僕は強いプログラミングが書けないので,お絵かき的なPietを描いて楽しんでいます.

2.2　Pietは凄い!

今現在様々な言語が世の中にはありますが, Pietは画像でプログラミングが出来るというとても異彩な言語です. 画像でプログラミングが出来るというのは大変すごいことです. 例えば,一見キャラクターの顔画像にしか見えないのに, Pietとして実行すると「72CHIHAYA」と出力し続けるものを描いたことがあります.

rubyが「プログラミングが楽しく出来るように」を，perlが「There’s more than one way to do it(やり方は一つじゃない)」を モットーにしていますが，Pietは画像でプログラミングが出来るので楽しく，上記の例のようにプログラミングを表現する絵は自分の好きなように描けますので，PietはRubyやPerlとおなじコンセプトを持った凄い言語と言えるでしょう!

2.3 僕の考えた最強のPiet (Piet#)

そんな素晴らしいPietですが,未だに難解プログラミング言語として扱われています．Piet自体には本当はもっと表現力があるのに…．今回の記事は,そんなPietを拡張してもっともっと実用的な 最強のPiet「Piet#」の仕様を考えたという記事です．

2.4 Piet#の新機能

- DLLのインポート
- 他のPietのコードの呼び出し
- スタックにスタックを積める
- 強力な文字処理機構
- 自分の好きな18色によるコーディング

2.5 Piet#で新たに出来ること

2.5.1 Webアプリケーションやゲームを簡単に作れる!

旧Pietでは標準入出力になにかをするので手一杯でした．しかし,Piet#では前述のDLLのインポートをサポートしているので，気軽に素晴らしいプログラミングができます! 更に,他のPietコードを呼び出せるので，作成した機能はモジュール化することができ，ライブラリを揃えれば上述のことはたやすく実現できます!

2.5.2 高い表現力により自由なコーディングが出来る!

旧Pietでは数字しかスタックに積めませんでした．そのため,とても表現力が低かったのです．しかし,Piet#では,新たにスタックにスタックを積むことが出来て，とても表現力豊かなプログラミングが出来ます．スタックにスタックに,…,スタックに,スタックを積むというふうに 無限にスタックにはスタックを積めるので，木構造のようなデータ構造も再現できます．また,新たにスタックに積んだスタックを 文字列とみなすことで高度な文字処理が出来る機構も搭載していて，ファイル処理や高度な文字処理にも耐えうるコーディングが可能となります．

2.5.3 楽しく直感的なプログラミングが出来る!

旧Pietでも楽しくプログラミングが出来たのですが, コメントの挿入も出来ず,決まった18色しか出来ないなどの成約がありました. しかし,Piet#では専用のIDEでそれらをカバーする予定です. また,Piet#のプログラミングで本質的に変わったことは, 新たにスタックが積めるようになったことだけなので, 学習コストも少なくて済みます! Pietがコードとして利用するPNG画像にはメタデータを挿入できることを利用して, コメントの挿入,好きな18色でのコーディングなどをサポートし, より快適なプログラミングを目指します!

2.6 PietとしてのPiet#

Piet#はあくまでもPietです. 色は18色しか扱わないし, 画像のみがあれば誰でも同じコードを再現できます. これらを守ることで,Cに対するC++のように, 旧Pietのコードもそのまま動かすことが可能となります.

2.7 色の問題

Piet標準色はチカチカするものばかり… しかも特に青系の色など,違いがわかりにくいものもあります. Piet#では,自分で18色を定義できます. とはいえ,好き勝手定義しては読みにくいですので,指針を. 色相(H)の順序付けが出来て,明度(彩度)の順序付けができればいいですよね. 6色なので,色相は基本的には60度程度違えばよいですね. 色を見分けられるように,30度以上の差を全てが保つ色にしましょう. 明度と彩度でSVの三角形は,色が分かるように, $0.5 < V < 1$,$0.2 < S <= 1$であるという条件のもと, V,Sを変えればすきな組み合わせに出来ますね.

2.8 Piet#としての情報の保存方法

Piet#(Pietも!)が実行するコードはPNGファイルです. 実はPNGにはメタデータを入れることが出来ます. Piet#のIDEではコメントを挿入することが出来ます. また,前述のとおり18色の指定も出来ます. そのほかにも,IDEとして保存しておきたい情報は数多くあります. それも全てこのメタデータに保存すればよいですね.

2.9 記法の定義

今までの項でPiet#の新たな機能を見ました. 以降の項でPiet#の具体的な仕様を見ていくことにしましょう. まずは記法の定義から見ていきましょう. DirectionPointerをDPと略します. CodelChooserをCCと略します.

空のスタックに順番に, 1 をPush, 2 をPush, 3 をPush, 4 をPush した時, スタックの変化を [] → [1] → [1,2] → [1,2,3] → [1,2,3,4] と表記します. つまり,一番右にあるものが一番最近Pushしたものとなります.

また, [1,2,3,4] にAdd命令を行うことを, Add [1,2,3,4] → [1,2,7] と表記できます. この場合,1,2 は直接演算に関係ないので, Add […,3,4] → […,7] のように記すことができます. さら

に演算に関係ない項目 … を省略し,以降の文章では Add [3,4] → [7] のように記します. 例えば,Sub[7,2] → [5] は,7 - 2 = 5 をしたことを意味しますね.

また,Piet#では前述のとおり数字とスタックのみで構成されています. しかし,文字処理のために糖衣構文を定義しておきます. まず,65 =“A”; 66 =“B”;67 =“C”;とUnicodeで対応しています. 文字処理のために [65,66,67] と積んだ場合, その数字が何を指しているかで表現したほうが見やすいので, [65,66,67] = [A,B,C] と表記しましょう. さらにもっと省略して,[A,B,C] = “ABC” という糖衣構文も定義します. Piet# では数字とスタックしか使わないということは大事なことなので忘れないでください.

2.10 モジュール呼び出しとスタック型の導入

Piet#では新たに2つの命令を導入します. 既存のPietと競合しないように導入しなければなりません. そこで,普通しない命令である 0除算命令,0剰余命令に 新たな命令としての意味を持たせるという形で導入します.

モジュール呼び出し Div [n,0] → []
n.png という名前のPietを読み込んで実行します.
現在の標準入力,標準出力,スタックのまま, そしてDP は 0 ,CC は 1 で初期化されて そのPietを一番左上の位置から実行します. 終了後,退避されていたDP,CCは元に戻ります. この命令によりファイルをモジュール化でき, ファイルの分割が可能になります. また,言語としての強さの要素である標準ライブラリを搭載し 簡単にこの命令でアクセス出来るように, nが負の場合は標準ライブラリから探すというふうにする予定です.

スタックの新規作成 Mod […,x1,x2,…,xn,n,0] → […,[x1,x2,…,xn]]
深さnまでの要素を積んだ新たなスタックをPushします
この関数により,Piet#のスタックに, 新たなスタックを積むことが出来るようになります
例: Mod […,C,H,Y,3,0] → […,[C,H,Y]]

2.11 スタック演算

今回の拡張によって数字だけでなくスタックも積めるようになりました. スタック同士の演算を新たに定義しましょう. スタックとスタック,数字と数字との演算なら問題はないのですが, スタックと数字が混じっていた場合についてはちょっと考えないといけません. 結論としては,その数字を一つのみを積んだ新たなスタックを作成して, それを利用してスタックとスタックの演算として扱います.

例えば,加算の関数 Add は, Add […,72,28] → […,100] と,数字同士なら演算されますが, スタック同士に Add を作用させると結合の意味になります.
例:) Add […,[A,B,C],[X,Y]] → […,[A,B,C,X,Y]]
そして,数字とスタックの演算の場合は, その数字を一つのみを積んだ新たなスタックを作成して, それを利用してスタックとスタックの演算として扱うので, Add […,C,[H,Y]] → Add […,[C],[H,Y]] → […,[C,H,Y]] と計算されます.

それでは早速命令を見ていきましょう!

2.11.1 ゼロ引数関数

Push , In(char) , In (number) はスタックとしては定義されません.

2.11.2 一引数関数

スタックの先頭から一つとり,それに演算を適用します.

Switch: 平坦化する (Flatten) 型 :: […] → […]
スタックにスタックが積まれて,更にそれにスタックが…という状態に 陥いったときにそれを全て平坦化してくれます. 例を見てみますよ.

- Switch […,[A,[B,C,[D],E],F,G]] → […,[A,B,C,D,E,F,G]]
- Switch […,[A,B,C]] → […,[A,B,C]]
- Switch […,[A]] → […,[A]]
- Switch […,[]] → […,[]]

Point : スタックの先頭を取得する 型 :: […,T] → […], T
スタックの先頭要素を一つとりだしてPushします.例をあげましょう.

- Point […,[A,B,C]] → […,[A,B],C]
- Point […,[A,B,C,[X,Y]]] → […,[A,B,C],[X,Y]]
- Point [[[Xn,Yn],⋯,[X1,Y1]]] → [[[Xn,Yn],⋯,[X2,Y2]],[X1,Y1]]
- Point […,[A]] → […,[],A]
- Point [[]] → []

Dup: コピーする (Duplicate) 型 :: […] → […],[…] 数字版のDupと同じです.例を見せます.

- Dup […,[[1,3,4],4,32]] → […,[[1,3,4],4,32],[[1,3,4],4,32]]

Not: スタックの中身が空の時 1 を,それ以外では 0 を返す 型 :: […] → Num
数字版のNotと意味的には一緒です. NotとSwitchを組み合わせることで条件分岐に使用出来ますね.

- Not [“Hello”] → [0]
- Not [[A]] → [0]
- Not [[]] → [1]

Pop: 捨てる　型 :: […] → []
数字版のPopと一緒です.例を一応載せときます.

- Pop [[A,B,C]] → []

Out(Char) : 文字列として出力する　型 :: […] → []
Switchで平坦化して文字列と見なせるようにしてから出力します.

- Out(Char) [[C,H,Y]] → [] : “CHY”と出力する
- Out(Char) [[千,早]] → [] : “千早”と出力する
- Out(Char) [[C,[H,I,[H,A],Y],A]] → [] : “CHIHAYA”と出力する
- Out(Char) [[C]] → [] : “C” と出力する.
- Out(Char) [[]] → [] : 何も出力しない.

Out(Num) : スタックの中身を全て取り出し,先頭に要素数を積む　型 :: [Tn,…,T1] → Tn,…,T1,Num
Out(Num) という名前ですが出力はしませんので注意して下さい. Mod0命令の逆演算となります.

- Out(Num) […,[A,B,C]] → […,A,B,C,3]
- Out(Num) […,[A,[B,C],D]] → […,A,[B,C],D,3]
- Out(Num) […,[A]] → […,A,1]
- Out(Num) […,[]] → […,0]

Div 0: モジュール呼び出し　Div 0 の 被除数が スタックの場合,それをSwitchで平坦化して文字列にし,その文字列のパスにあるPietモジュールを実行します.

Mod 0 : スタックの新規作成　Mod […,x1,x2,…,xn,n,0] → […,[x1,x2,…,xn]]
深さnまでの要素を積んだ新たなスタックをPushします. x1,…,xn がスタック型であっても大丈夫です. n がスタックであった場合は n = ∞ として(つまり深さ無限で)演算します.

- Mod […,C,H,Y,3,0] → […,[C,H,Y]]
- Mod […,[C,H,I],[H,A,Y,A],2,0] → […,[[C,H,I],[H,A,Y,A]]]
- Mod […,C,H,Y,[],0] → [[…,C,H,Y]]

2.11.3　二引数関数

スタックの先頭から2つとり,それらに演算を適用します.

Add: 結合する (Append)　型 :: […],[…] → […]
単純に結合します.例を示します. [A,B,C] =“ABC” という糖衣構文を思い出してくださいね.

- Add […,[C,H,I],[h,a,y,a]] → […,[C,H,I,h,a,y,a]]
- Add […,“CHIhaya”,“isGOD”) → […,“CHIhayaisGOD”]
- Add […,1,[10,100,1000]] → […,[1,10,100,1000]]
- Add […,[1,10,100],1000] → […,[1,10,100,1000]]
- Add […,[A,B,C],[]] → […,[A,B,C]]
- Add […,[],[]] → […,[]]

Sub: 正規表現で分割する (Split)　型 :: […],[…] → […]
Switch で平坦化し,文字列として見なせるようにしてからSplitします. [A,B,C] =“ABC” という糖衣構文を思い出してくださいね.

- Sub […,“Pietはx楽しい”,“x”]→ […,[“Pietは”,“楽しい”]]
- Sub [“12/12 21:41”,” |/|:“] → [[”12“,”12“,”21“,”41”]]
- Sub […,“helloHWWH!”,H] → […,[“hello”,[W,W],[!]]]
- Sub […,[h,[e,H],[l,l,[[o]]]],[H] → […,[“he”,“llo”]]
- Sub […,H,“
  s”] → […,[H]]
- Sub […,H,“H”] → […,[]]

Mul: 直積集合を取得する　型 :: […],[…] → […]
便利な未来を作る直積集合ですよ.

- Mul [[A,B,C],[X,Y]] → [[[A,X],[A,Y],[B,X],[B,Y],[C,X],[C,Y]]]
- Mul […,H,[J,I]] → […, [[H,J],[H,I]]]
- Mul […,[A],H] → […,[A,H]]
- Mul […,[],A]→ […,[]]
- Mul [[A,B,C],[[X,Y]]] → [[A,[X,Y]],[B,[X,Y]],[C,[X,Y]]]
- これにより,例えば $0 < n,m \leq 20$ の $n * m$ の取りうる値を列挙できます
- 1～20までと20と0を順に積む → [1,…,20,20,0]
- Mod0 を実行してスタック化 → [[1,…,20]]

- Dupでコピーを作成 → [[1,…,20],[1,…20]]
- Mul で直積を取る → [[[1,1],…,[20,20]]]
- Point で先頭の要素を取得 → [[[1,1],…,[20,19]],[20,20]]
- Out(N)で分解する → [[[1,1],…,[20,19]],20,20,2]
- 2 を Pop → [[[1,1],…,[20,19]],20,20]
- Mulで掛け算する → [[[1,1],…,[20,19]],400]
- Roll 2 1 で入れ替える → [400,[[1,1],…,[20,19]]]
- 5~9 を繰り返す.
- [400,…,1] が出来上がる.

Div: 正規表現でマッチさせる (Match)　型 :: […],[…] → […]
Switch で平坦化し,文字列として見なせるようにしてからMatchします. [A,B,C] =“ABC” という糖衣構文を思い出してくださいね.

- Div [“aa9bb1cc”,“.[1-9].”] → [[“a9b”,“b1c”,“aa9bb1cc”]]
- Div [“aa”,“.[1-9].”] → [[]]
- Div [“aHxH”,H]→ [[“H”,“H”,“aHxH”]]
- Div [H,“h|H”] → [[“H”,“H”]]
- Div [[],“A|B”] → [[]]
- Div [“ABC”,[]] → [[]]

Mod: スタックを順番に結合する (Combine)　型 :: […],[…] → […]
それぞれの要素を結合します.Mulと上手く組み合わせてご使用下さい.

- Mod [“01234”,“abc”] → [[“2a”,“3b”,“4c”]]
- Mod [[a,b,c,d,e],[x,y]] → [[[d,x],[e,y]]]
- Mod [[[a,b,c],[d,e]],[x,y]] → [[[[a,b,c],x],[[d,e],y]]]
- Mod [H,“abc”]→ [[H,c]]
- Mod [“abc”,H]→ [[c,H]]
- Mod [“abc”,[]]→ [[]]
- Mod [[],[]]→ [[]]

Greater: ファイルを読み書きする　型 :: […],[…] → Num
ファイルを開いて標準入出力へリダイレクトします. 成功したら 1 ,失敗したら 0 が積まれます.
第一引数は開くファイル名です. 第二引数は開くモードを指定します.
モード

- [R] or [r] or [0] : 読み込み(Read)モードで開く.
- [W] or [w] or [1] : 書き込み(Write)モードで開く.
- [A] or [a] or [2] : 追加書き込み(Append)モードで開く
- [CR] or [cr] or [3] : Readで開いたファイルを閉じる.
- [Cw] or [cw] or [4] : Writeで開いたファイルを閉じる.
- その他 : 現在開いているファイルをRead,Writeともに閉じる.

新たにファイルを開いた場合,現在開いているファイルは閉じられます. 例を示します.

- Greater [“f1.txt”,r] → [1] : f1.txtを開き標準入力をリダイレクトする
- Greater [“f2.txt”,1] → [1] : f2.txtを開き標準出力をリダイレクトする
- Greater [“f.t”,r] → [0] : f.tが開けなかった.
- Greater [“f3.txt”,a] → [1] 2.で開いていたf2.txtを閉じて新たにf3.txtを
  追加書き込みモードで開き,標準出力をリダイレクトする.
- Greater [[],[cr]] → [0] : 1.で開いていたf1.txtを閉じる.

2.11.4　四引数関数

2.11.5　Roll: Dllを読み込む

型 :: […],[…],[…],[…] → Num or […] or Empty
Pietが何でも出来るようになる魔法の関数です. この関数により,ゲームやアプリを作ることが出来るようになります. 実用的なマルチメディアな操作をPietで体感しよう!
引数
第一引数 : DLLへのファイル名を含むパス
第二引数 : DLL内の関数名
第三引数 : DLLに渡される型の宣言
第四引数 : DLLを呼び出すための引数リスト
例

- Roll [“user32.dll”,“MessageBoxA”, “vssui”, [[],“Hello World.”,“Caption”,0]]
  C製Dllであるuser32.dllの,MessageBoxA関数を呼び出します. “vssui”というのは的確にDLLを扱う為の型情報の宣言です. 引数の型は前から順に void*(HANDLE),char(LPSTR), char*(LPSTR),unsigned int (UINT) で, 返り値はint型にするという宣言で, 前から順にvoid*,char*,char*,unsigned int,intなので 頭文字をとって “vssui”です.

- Roll [“TestDLL.dll”,“TestDll.CsDll.LangA”,0,[123,“RA”]]
  自作.Net製DllでTestDll.CsDll.LangA(123,“RA”)を実行します. 第二引数は NAMESPACE.CLASSNAME.STATICFUNCTIONNAME とします. .Net製Dllは型情報が内蔵されているために, 型を推論することができ,第三引数は省略できます. というか,型を書いても無視されます. C製Dllで扱えるプリミティブな型以外はサポートしません.

DLL読み込みの型 Pietの最強のIDEであるPidetがC#製なので Piet#のIDE,実行ファイルもC#製です. そのため,内部的にはC#で処理します. 型の相互変換表は下のようになります.

Pietでの型	Cでの型	Cでの別名	C#での型
Number	int,long	INT,LONG,BOOL	int
Number	unsigned int,unsigned long	UINT,DWORD,ULONG	uint
Number	unsigned char	BYTE	byte
Number	char	CHAR	char
Number	unsigned short	WORD	ushort
Number	short	SHORT	short
Number	float	FLOAT	float
Stack	char*	LPSTR,LPWSTR	string
Stack	void*	HANDLE	IntPtr

2.12 実装

ここまで,仕様を詳細に示してきました. あとは実装するだけですね! C#で書かれたPidetがあるのでそれをforkして作りたいと思います. Piet#.exe を実行すれば出来るようにしたいですね. スクリプト言語としてのPietとなって, Perl,PHP,Python,Piet# の 4Pとなることを願っています. このPiet#プロジェクトは,ここまでの基本仕様が出来た時点で公開します. オープンソースプロジェクトとして, 公開後は標準ライブラリの実装のContributeをお待ちしております! 最新情報は Github : Muratam, Twitter : paradigm_9 を参照下さい!

Chapter 3

あとがきの国

NoNameA 774　今回初めて本を出すことに挑戦したのですが、動き出すのが遅れて大変でした……。こんな本でしたがここまで読んでいただいてくれてるのなら本当にありがとうございました。

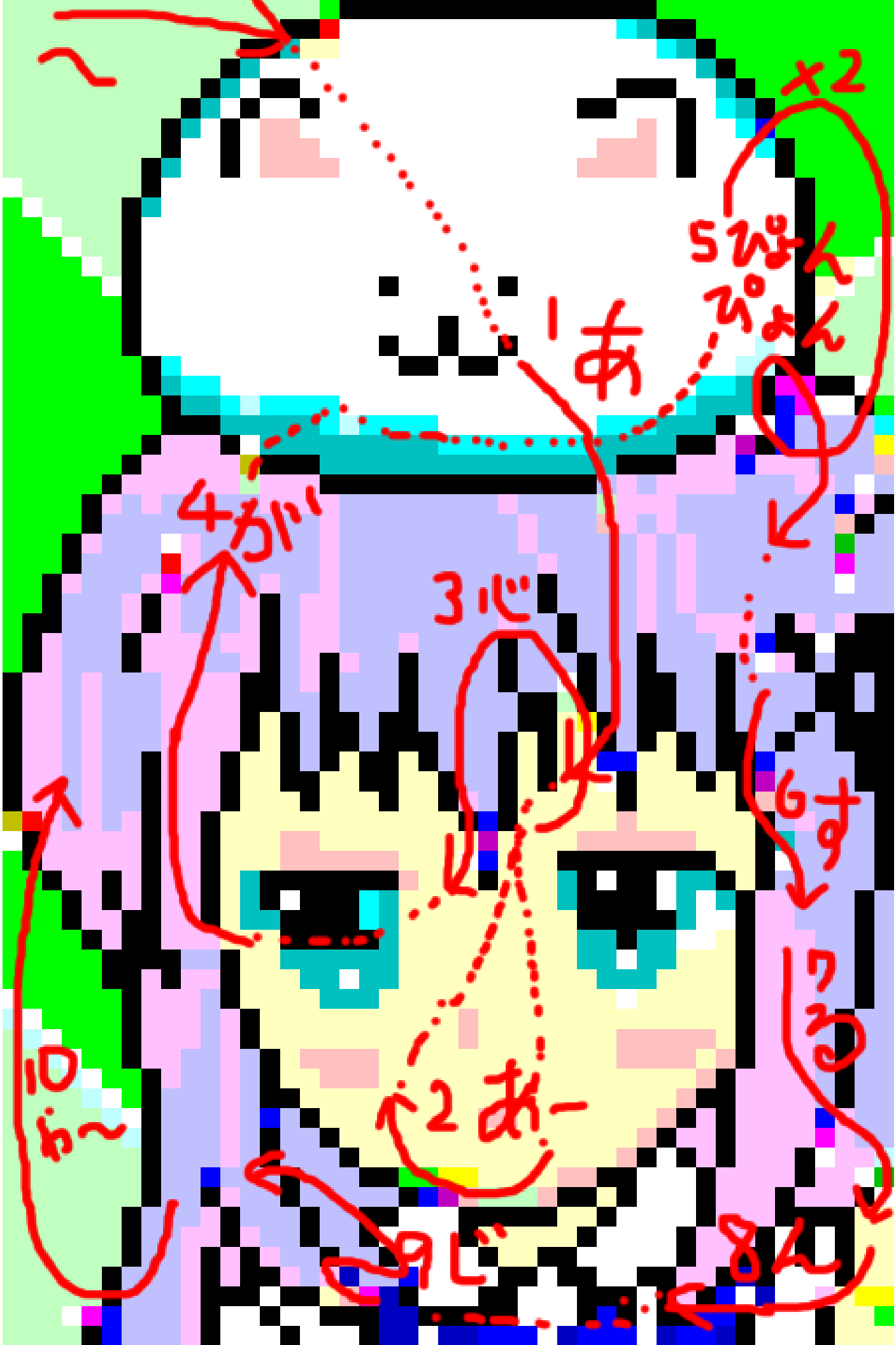
murata　神はまずPietを作られた。世界は救われた。「ピエト福音書」より

PDF配布について　今回値段の関係からカラーでの頒布を諦めたので(Pietは画像を扱うプログラムである以上、ぜひカラーにしたかったのですが……)、カラフルなPDFを配布させて頂きます。以下のURIから、user:passはpiet:automataでダウンロードお願いします。
https://nna774.net/piet/C89Book.pdf
　ぜひこのPDFをあなたの友人へと自由に共有してください。

奥付

2015/12/31	初版発行
hash:	3e0c955600ecae2e6549ea95b9c9eee7587ef01c(の次)
著作・発行	NoNameA 774 (nonamea774@nnn77)
サークル	いっと☆わーくす!
メールアドレス	nonamea774@gmail.com
Web	https://nna774.net/
Twitter	@nonamea774
GPG Key	0x0C3E3AB2
fingerprint	674A 287A 21D2 2431 AD8F D328 AEF3 C3C7 0C3E 3AB2

　表紙のPietはmurataによる”「ああ～心がぴょんぴょんするんじゃ～」と出力するチノちゃん”です。https://twitter.com/paradigm_9/status/677612736063258624にて入手出来ます。


1あ
2あー
3じ
4が
5ぴょん
×2
6す
7ろ
8ん
9じ
10ら〜