PostScript 実習マニュアル

version 1.03
2003 年 1 月 24 日（金）

1 PostScript の基礎知識

1.1 PostScript とは何か

この文章は、PostScript というものについて解説することを主要な目的として書かれたものです。そこで、まず最初に、そもそも PostScript というのはいったい何なのか、ということについて説明しておくことにしましょう。

ページの上に印刷される文字や図形や画像などを記述するための言語のことを「ページ記述言語」(page description language) と言います。PostScript というのは、そのようなページ記述言語のひとつで、Adobe Systems Incorporated という会社によって開発されたものです。

PostScript には、ほかのページ記述言語にはないユニークな特徴があります。それは、プログラミング言語としての機能を持っているということです。つまり、PostScript には、C や Pascal や Lisp や Ruby などと同じように、数値の計算をしたり、動作を繰り返したり、動作の選択をしたりする、ということを記述する能力があるのです。

1.2 Ghostscript

人間が書いたプログラムを実行するためには、「言語処理系」(language precessor) と呼ばれるプログラムを使う必要があります。言語処理系には、「コンパイラ」(compiler) と「インタプリタ」(interpreter) という二つの種類があるのですが、PostScript で書かれたプログラムは、普通、インタプリタによって実行されます。

パソコンの上で動く PostScript のインタプリタとしては、Ghostscript という名前のプログラムが、もっともよく使われています。Linux の場合、Ghostscript は、

```
gs
```

というコマンドをシェルに入力することによって起動することができます。

Ghostscript は、起動すると、

```
GS>
```

というプロンプトを出力します。このプロンプトが出ているときに PostScript のプログラムを入力すると、そのプログラムが即座に実行されます。たとえば、

```
437 ==
```

というプログラムを入力してエンターキーを押すと、437 という整数が出力されて、そののち、ふたたびプロンプトが出力されます。

Ghostscript を終了させたいときは、

```
quit
```

と入力します。

1.3 スタック

PostScript で書かれたプログラムは、「スタック」(stack) と呼ばれるものを使うことによって実行されます。スタックというのは、その中にデータを一列に並べておくことのできる容器の一種です。スタックにデータを入れることを、データを「プッシュする」(push) と言い、スタックからデータを取り出すことを、データを「ポップする」(pop) と言います。スタックという容器

の特徴は、データの出し入れをするための出入り口が一方の端にしかない、というところにあります。つまり、スタックからデータをポップすると、いちばん最後にプッシュしたデータが取り出される、ということになります。

PostScript のプログラムは、「トークン」(token) と呼ばれる単語が一列に並ぶことによってできています。たとえば、

```
437 ==
```

というプログラムは、437 と == という二つのトークンから構成されています。

PostScript のインタプリタは、プログラムの中のトークンを、左から右へ順番に処理していきます。上のプログラムの場合、PostScript のインタプリタは、まず 437 というトークンを処理して、次に == というトークンを処理します。

数値というのは、トークンの一種です。インタプリタによる数値の処理は、ただ単に、それをスタックにプッシュするだけです。たとえば、

```
47 81 -53 6.027
```

というプログラムをインタプリタに実行させると、インタプリタは、その中の数値を、左にあるものから順番にスタックにプッシュしていきます。ですから、このプログラムの実行が終了した時点で、スタックの中には、

```
47 81 -53 6.027
```

というように数値が並ぶことになります（右端がスタックの出入口だと考えてください）。

1.4 オペレーター

PostScript のインタプリタの中には、「オペレーター」(operator) と呼ばれるものがたくさん組み込まれています。それぞれのオペレーターは、それぞれに異なった何らかの動作を実行します。オペレーターにはかならず名前が付いていて、オペレーターの名前も、「オペレーター」という言葉で呼ばれます。オペレーター（の名前）は、プログラムのトークンになります。インタプリタにオペレーターを処理させると、インタプリタは、そのオペレーターを動作させます。

最初にインタプリタに実行させてみた、

```
437 ==
```

というプログラムの中で使われている == というのも、オペレーターのひとつです。この == というオペレーターは、スタックから 1 個のデータをポップして、そのデータを出力する、という動作をします。ですから、このプログラムをインタプリタに実行させると、インタプリタは、まず 437 をスタックにプッシュして、そして次に == というオペレーターを動作させます。そうすると、 == がスタックから 437 をポップして、それをモニターに出力するわけです。

pstack というのも、 == と同様に、データをモニターに出力するオペレーターです。ただし、== とは違って、pstack は、スタックからデータをひとつもポップしないで、スタックの内容をすべて出力します。

それでは、

```
-603 291 522 433 pstack pstack
```

というプログラムをインタプリタに実行させてみてください。

1.5 算術オペレーター

PostScript のインタプリタに含まれているオペレーターのうちで、加減乗除などの算術演算を実行するオペレーターは、「算術オペレーター」(arithmetic operator) と呼ばれます。たとえば、算術オペレーターのひとつで add というオペレーターがあるのですが、これは 2 個の数値を加算するという動作をします。

算術オペレーターを使いたいときは、まず演算の対象となる数値をスタックにプッシュして、それから算術オペレーターを実行します。そうすると、算術オペレーターは、演算の対象となる数値をスタックからポップして、それから演算を実行して、そして演算の結果をスタックにプッ

a b add	a と b とを加算します。
a b sub	a から b を減算します。
a b mul	a と b とを乗算します。
a b div	a を b で除算したときの商を求めます。
a b idiv	整数の範囲で、a を b で除算したときの商を求めます。
a b mod	a を b で除算したときのあまりを求めます。
a neg	a の符号（プラスかマイナスか）を反転させます。

表 1: 算術オペレーター

シュします。たとえば、25 と 43 を加算したいならば、まず 25 と 43 をスタックにプッシュして、それから add を実行します。そうすると、 add が 43 と 25 をスタックからポップして、それらを加算して、その結果 (68) をスタックにプッシュします。ですから、そのあとで == を実行すると、演算の結果が出力されることになります。

それでは、

```
25 43 add ==
```

というプログラムをインタプリタに実行させてみてください。そうすると、25 と 43 を加算した結果、つまり 68 が出力されるはずです。

表 1 は、いくつかの算術オペレーターについて、それらがどのような動作をするかということを表の形にまとめたものです。この表の中の a と b は、a が先にプッシュしたデータ、b があとからプッシュしたデータをあらわしています。

演算の結果に対して別の演算を実行したいときは、演算の順序のとおりに数値やオペレーターを並べて書きます。

```
31 27 sub 100 mul neg ==
```

このプログラムをインタプリタに実行させると、まず 31 から 27 が減算されて、その結果 (4) と 100 とが乗算されて、その結果 (400) の符号が反転されて、その結果 (−400) が出力されます。

1.6 スタックオペレーター

PostScript には、スタックの内容を操作するためのさまざまなオペレーターが組み込まれています。そのようなオペレーターは、「スタックオペレーター」(stack operator) と呼ばれます。スタックオペレーターとしては、次のようなものがあります。

`clear` スタックの内容をすべて削除します。

`pop` スタックからひとつのデータをポップします。ポップしたデータは、なくなってしまいます。

`exch` スタックの先頭と 2 番目とで、それぞれの位置のデータを入れ替えます。

n j `roll` スタックの先頭から n 個のデータを、j 回だけ回転させます。回転の方向は、j がプラスの場合は奥から手前、j がマイナスの場合は手前から奥です。

`dup` スタックの先頭（スタックの出入り口にもっとも近い場所）にあるデータの複製を作って、それをスタックにプッシュします。

n `copy` スタックの先頭から n 個のデータの複製を作って、それらをスタックにプッシュします。

n `index` スタックの先頭から数えて n 番目のデータ（先頭を 0 番目と数えます）の複製を作って、それをスタックにプッシュします。

1.7 ファイルに格納されたプログラムの実行

PostScript のプログラムをファイルに格納しておいて、そのファイルの内容をインタプリタに実行させる、ということも可能です。ただし、PostScript のプログラムをファイルに格納する場合は、そのプログラムの先頭に、

```
%!PS-Adobe-3.0
```

という記述を書いておく必要があります。これは、自分の下に書かれているものが PostScript のプログラムだということをインタプリタに知らせる、という役割を持っている記述です。

それでは、実際に試してみましょう。まず、 output.ps という名前のファイルに次のプログラムを格納してください。ちなみに、PostScript のプログラムをファイルに格納する場合、そのファイルの名前には .ps という拡張子を付けることになっています。

```
%!PS-Adobe-3.0
437 ==
```

ファイルに格納されているプログラムをインタプリタに実行させたいときは、インタプリタに対して、

(ファイル名) run

というプログラムを入力します。ですから、

```
(output.ps) run
```

というプログラムを入力すると、 output.ps に格納したプログラムが実行されるはずです。

2 直線

2.1 線を描画する手順

PostScript というのはページ記述言語ですから、それを使うことによって、ページの上にさまざまなグラフィックスを描画するプログラムを書くことができます。そこで、まず手始めに、直線を描画するプログラムを書く方法について説明していきたいと思います。

PostScript では、直線や曲線などの線が組み合わさってできている図形のことを「パス」(path) と呼びます。直線や曲線を描画したいときは、まず、描画したい直線や曲線から構成されるパスを作って、そののちそれを描画する、という 2 段階の操作をする必要があります。

パスを作りたいときは、 newpath というオペレーターを使います。 newpath を実行すると、空のパスが新しく作られて、そこに線を追加することができる状態になります。ちなみに、線を追加することができる状態になっているパスは、「カレントパス」(current path) と呼ばれます。

パスを描画したいときは、 stroke というオペレーターを使います。 stroke を実行すると、カレントパスが描画されて、そののち、カレントパスは空の状態に戻ります。

stroke がその上にパスを描画する平面は、「カレントページ」(current page) と呼ばれる仮想的な平面ですので、その段階ではまだ人間の目には見えません。カレントページに描画されているものをモニターやプリンターなどの出力装置に送るためには、 showpage というオペレーターを実行する必要があります。

以上の手順を整理してみると、PostScript を使って直線や曲線を描画したいときは、

(1) newpath で新しいパスを作る。
(2) カレントパスに線を追加する。
(3) stroke でカレントパスをカレントページに描画する。
(4) showpage でカレントページを出力装置へ転送する。

という手順を実行すればいい、ということになります。

2.2 カレントポイント

それでは次に、カレントパスに直線を追加する方法について説明しましょう。

カレントパスに直線を追加するためには、その直線の位置を指定する必要があります。PostScript では、ページの上の位置を、x 軸と y 軸から構成される座標系を使って指定します。

座標系は、PostScript のオペレーターを使って、かなり自由に設定することができるのですが、デフォルトの状態では、原点（つまり x 軸と y 軸とが交わっている点）がカレントページの左下の隅にあって、x 軸はそこから右向きに延びていて、y 軸はそこから上向きに延びています。

PostScript では、直線や曲線は、ひとつの仮想的なペンによって作られます。そのペンが存在する位置は、「カレントポイント」(current point) と呼ばれます。`newpath` でパスが作られた直後には、カレントポイントは、まだどこにも存在していません。カレントパスに直線を追加したいときは、まず最初に、`moveto` というオペレーターを使ってどこかにカレントポイントを作る必要があります。`moveto` を使うときは、

x y `moveto`

というように、あらかじめ 2 個の数値をスタックにプッシュしておく必要があります。x は、カレントポイントにしたい位置の x 座標で、y は y 座標です。数値の単位としては、印刷物を作るときによく使われる、「ポイント」(point) と呼ばれる単位を使います。1 ポイントというのは、72 分の 1 インチ、つまり約 0.35 ミリに相当します。たとえば、

```
200 300 moveto
```

を実行すると、(200,300) という座標であらわされる位置がカレントポイントになります。

カレントパスに直線を追加したいときは、`lineto` というオペレーターを使います。`lineto` は、スタックから y 座標と x 座標をポップして、カレントポイントと (x, y) とをつなぐ直線をカレントパスに追加して、カレントポイントを (x, y) に移動させます。たとえば、

```
400 500 lineto
```

を実行すると、カレントポイントと (400,500) とをつなぐ直線がカレントパスに追加されて、カレントポイントが (400,500) に移動します。

プログラムの例 `lineto.ps`

```
%!PS-Adobe-3.0
newpath
100 200 moveto
400 200 lineto
250 400 lineto
stroke
showpage
```

一筆書きでは描けない図形を作るためには、カレントパスに線を追加しないでカレントポイントを移動させる必要があります。そのようなときは、カレントポイントを作るのに使った `moveto` というオペレーターを使います。`moveto` は、すでにカレントポイントが存在する場合は、ただ単に、指定された位置へカレントポイントを移動させるという動作をします。

プログラムの例 `batsu1.ps`

```
%!PS-Adobe-3.0
newpath
100 200 moveto
200 300 lineto
100 300 moveto
200 200 lineto
stroke
showpage
```

2.3 相対的な位置指定

moveto と lineto は、スタックからポップした数値を絶対的な座標だと解釈するわけですが、ポップした数値を、カレントポイントに対する相対的な位置だと解釈するオペレーターもあります。それは、 rmoveto と rlineto というオペレーターです。

x y rmoveto

は、カレントポイントを出発点として、右へ x、上へ y だけ移動した位置へカレントポイントを移動させます。左や下へ移動させたいときは、マイナスの数値を指定します。同じように、

x y rlineto

は、カレントポイントと、そこから右へ x、上へ y だけ移動した位置とをつなぐ直線をカレントパスに追加して、そののち、カレントポイントを移動させます。

プログラムの例 batsu2.ps

```
%!PS-Adobe-3.0
newpath
100 200 moveto
100 100 rlineto
-100 0 rmoveto
100 -100 rlineto
stroke
showpage
```

なお、 rmoveto にはカレントポイントを作るという機能がありませんので、カレントポイントがまだ存在していないときは、 rmoveto を使うことができません。

2.4 線の幅

stroke を使ってカレントパスをカレントページに描画するときの線の幅は、デフォルトでは 1 ポイントになっています。線の幅を設定したいときは、 setlinewidth というオペレーターを使います。 setlinewidth は、スタックから 1 個の数値をポップして、それを線の幅として設定します。

プログラムの例 shikaku.ps

```
%!PS-Adobe-3.0
newpath
200 200 moveto
200 0 rlineto
0 200 rlineto
-200 0 rlineto
0 -200 rlineto
70 setlinewidth
stroke
showpage
```

このプログラムで描画される正方形は、左下の頂点のところが少し欠けています。これは、図形の開始点と終了点とが接続されていないためです。図形の開始点と終了点とを接続したいときは、最後の直線をカレントパスに追加するときに、 lineto でも rlineto でもなく、 closepath というオペレーターを使います。

プログラムの例 close.ps

```
%!PS-Adobe-3.0
newpath
200 200 moveto
200 0 rlineto
0 200 rlineto
-200 0 rlineto
closepath
70 setlinewidth
stroke
showpage
```

2.5 色

stroke によって描画される線の色は、デフォルトでは黒色になっています。カレントパスを描画するときの色を設定したいときは、setrgbcolor というオペレーターを使います。setrgbcolor を使うためには、あらかじめ 3 個の数値をスタックにプッシュしておく必要があります。setrgbcolor は、それらの数値を、プッシュされた順番に、赤、緑、青、という光の三原色の比率だとみなして、それらの原色を混ぜ合わせることによってできる色を設定します。

プログラムの例 color.ps

```
%!PS-Adobe-3.0
newpath
30 setlinewidth
100 660 moveto 400 0 rlineto 1   0   0   setrgbcolor stroke
100 620 moveto 400 0 rlineto 0   1   0   setrgbcolor stroke
100 580 moveto 400 0 rlineto 0   0   1   setrgbcolor stroke
100 540 moveto 400 0 rlineto 1   1   0   setrgbcolor stroke
100 500 moveto 400 0 rlineto 0   1   1   setrgbcolor stroke
100 460 moveto 400 0 rlineto 1   0   1   setrgbcolor stroke
100 420 moveto 400 0 rlineto 0   0.5 1   setrgbcolor stroke
100 380 moveto 400 0 rlineto 1   0.5 0   setrgbcolor stroke
100 340 moveto 400 0 rlineto 0.5 0.5 1   setrgbcolor stroke
100 300 moveto 400 0 rlineto 1   0.5 0.5 setrgbcolor stroke
100 260 moveto 400 0 rlineto 0.5 0   0   setrgbcolor stroke
100 220 moveto 400 0 rlineto 0   0.5 0   setrgbcolor stroke
100 180 moveto 400 0 rlineto 0   0   0.5 setrgbcolor stroke
100 140 moveto 400 0 rlineto 0.6 0.6 0.6 setrgbcolor stroke
100 100 moveto 400 0 rlineto 0.9 0.9 0.9 setrgbcolor stroke
showpage
```

2.6 ラインキャップ

線が開始したり終了したりするところの形状のことを「ラインキャップ」(line cap) と言います。ラインキャップには、バット (batt)、ラウンド (round)、プロジェクティングスクエア (projecting square) という 3 種類のものがあります。ラインキャップを変更したいときは、setlinecap というオペレーターを使います。あらかじめ、0、1、2、のいずれかの整数をスタックにプッシュしておいてから setlinecap を実行すると、ラインキャップは、整数が 0 ならばバット、1 ならばラウンド、2 ならばプロジェクティングスクエアに変更されます。ラインキャップのデフォルトは、0 のバットです。

プログラムの例 cap.ps

```
%!PS-Adobe-3.0
newpath
70 setlinewidth 0.5 1 1 setrgbcolor
150 400 moveto 450 400 lineto 0 setlinecap stroke
150 300 moveto 450 300 lineto 1 setlinecap stroke
150 200 moveto 450 200 lineto 2 setlinecap stroke
150 400 moveto 450 400 lineto
150 300 moveto 450 300 lineto
150 200 moveto 450 200 lineto
1 setlinewidth 1 0 0 setrgbcolor 0 setlinecap stroke
showpage
```

2.7 ラインジョイン

2 本の線が連結されているところの形状のことを「ラインジョイン」(line join) と言います。ラインジョインには、マイター (miter)、ラウンド (round)、ベベル (bevel) という 3 種類のものがあります。ラインジョインを変更したいときは、setlinejoin というオペレーターを使います。あらかじめ、0、1、2、のいずれかの整数をスタックにプッシュしておいてから setlinecap を

実行すると、ラインジョインは、整数が 0 ならばマイター、1 ならばラウンド、2 ならばベベルに変更されます。ラインジョインのデフォルトは、0 のマイターです。

プログラムの例 join.ps

```
%!PS-Adobe-3.0
newpath
70 setlinewidth 0.5 1 1 setrgbcolor
150 600 moveto 400 600 lineto 300 700 lineto
0 setlinejoin stroke
150 400 moveto 400 400 lineto 300 500 lineto
1 setlinejoin stroke
150 200 moveto 400 200 lineto 300 300 lineto
2 setlinejoin stroke
150 600 moveto 400 600 lineto 300 700 lineto
150 400 moveto 400 400 lineto 300 500 lineto
150 200 moveto 400 200 lineto 300 300 lineto
1 setlinewidth 1 0 0 setrgbcolor 0 setlinecap stroke
showpage
```

2.8 破線

破線を描画したいときは、あらかじめ、その破線のパターンとオフセットというものを設定しておく必要があります。破線の「パターン」(pattern) というのは、破線の中の線の部分と間隔の部分の長さをあらわす数値の列のことです。パターンは、プログラムの中に、

[数値 数値 数値 ···]

という形のものを書くことによって作ることができます。この中の数値は、奇数番目が線の長さ、偶数番目が間隔の長さをあらわします。たとえば、

```
[30 20]
```

というパターンは、30 という長さの線と 20 という長さの間隔とを繰り返すような破線を作ります。また、

```
[50 10 20 10]
```

というパターンは、長い線と短い線とが交互に現われるような破線を作ります。

「オフセット」(offset) というのは、破線のパターンの中のどの位置から描画を開始するか、ということを指定する数値のことです。たとえば、オフセットとして 20 を指定したとすると、パターンの先頭から 20 だけ進んだ位置から破線の描画が開始されます。

破線のパターンとオフセットを設定したいときは、設定したいパターンとオフセットをスタックにプッシュしてから、setdash というオペレーターを実行します。たとえば、線が 40 で間隔が 20 というパターンと 0 というオフセットを設定したいならば、

```
[40 20] 0 setdash
```

と書けばいいわけです。

プログラムの例 dash.ps

```
%!PS-Adobe-3.0
newpath
5 setlinewidth 0 0 1 setrgbcolor
100 600 moveto 400 0 rlineto [30 20] 0 setdash stroke
100 500 moveto 400 0 rlineto [30 20] 20 setdash stroke
100 400 moveto 400 0 rlineto [50 10 20 10] 0 setdash stroke
100 300 moveto 400 0 rlineto [30] 0 setdash stroke
100 200 moveto 400 0 rlineto [30 20 10] 0 setdash stroke
100 100 moveto 400 0 rlineto [] 0 setdash stroke
showpage
```

ちなみに、破線の場合も、ラインキャップやラインジョインを変更することができます。

プログラムの例 roudash.ps

```
%!PS-Adobe-3.0
newpath
100 100 moveto 400 0 rlineto -200 500 rlineto closepath
40 setlinewidth 0 0 1 setrgbcolor [120 50] 0 setdash
1 setlinecap 1 setlinejoin stroke
showpage
```

3 円弧

3.1 中心の指定による円弧

カレントパスに円弧を追加する方法としては、中心を指定する方法と接線を指定する方法という 2 種類のものがあります。

中心を指定することによって円弧を作りたいときは、 arc または arcn というオペレーターを使います。 arc と arcn との相違点は、ペンを動かす方向が違うという点だけです。 arc は反時計回りでペンを動かし、 arcn は時計回りでペンを動かします。

arc または arcn を実行するときは、それに先立って、5 個の数値をスタックにプッシュしておく必要があります。その数値というのは、プッシュする順番で言うと、中心の x 座標、y 座標、半径、開始角度、終了角度です。角度は、円弧の中心を基準として右の方向が 0 度で、反時計回りで大きくなっていきます（角度の単位は度です）。

たとえば、プログラムの中に、

```
200 300 100 45 315 arc
```

と書いたとすると、中心の座標が (200,300)、半径が 100、開始角度が 45 度、終了角度が 315 度で、ペンを反時計回りで動かすことによってできる円弧がカレントパスに追加されます。

プログラムの例 arc.ps

```
%!PS-Adobe-3.0
20 setlinewidth
newpath
300 400 100 0 225 arc
stroke
300 200 100 0 225 arcn
stroke
showpage
```

カレントポイントが存在しているときに arc や arcn を実行すると、カレントパスに、まずカレントポイントと円弧の開始点とをつなぐ直線が追加されて、そののち円弧が追加されます。そして、 arc や arcn が実行されると、カレントポイントは、円弧の終了点へ移動します。

プログラムの例 kofun.ps

```
%!PS-Adobe-3.0
20 setlinewidth
newpath
100 100 moveto
200 0 rlineto
200 350 100 315 225 arc
closepath
stroke
showpage
```

3.2 接線の指定による円弧

直線を組み合わせてグラフィックスを作るときに、直線と直線とがつながっている部分を丸くしたい、という場合は、中心を指定して円弧を作るのではなくて、二本の接線を指定して円弧を作るほうが簡単です。

接線を指定することによって円弧を作りたいときは、arcto というオペレーターを使います。arcto を実行するときには、それに先立って、5 個の数値をスタックにプッシュしておく必要があります。その数値というのは、プッシュする順番で言うと、一本目の接線と二本目の接線とがつながる点の x 座標と y 座標、二本目の接線の終了点の x 座標と y 座標、そして半径です（一本目の接線の開始点は、カレントポイントです)。たとえば、プログラムの中に、

```
400 300 200 150 100 arcto
```

と書いたとすると、カレントポイントと (400,300) とをつなぐ直線と (400,300) と (200,150) とをつなぐ直線とを接線とする、半径が 100 の円弧が、カレントパスに追加されます。

arcto によって作られる円弧は、二つの接点のあいだだけです。カレントポイントは、arcto が実行されたのち、二本目の接線と円弧との接点に移動します。また、円弧だけではなく、カレントポイントから一本目の接線と円弧との接点までの直線も、カレントパスに追加されます。

arcto は、二つの接点の座標をスタックにプッシュして、それをスタックに残したまま終了します。それらの座標を使わない場合は、pop を 4 回実行することによってそれらをスタックから取り除く必要があります。

プログラムの例 arcto.ps

```
%!PS-Adobe-3.0
newpath
100 100 moveto
500 600 100 600 140 arcto
pop pop pop pop
20 setlinewidth 0.5 1 0.5 setrgbcolor stroke
100 100 moveto
500 600 lineto
100 600 lineto
2 setlinewidth 0 0 0.5 setrgbcolor stroke
showpage
```

4 ベジェ曲線

4.1 ベジェ曲線の基礎

パスには、直線と円弧を追加することができるわけですが、それだけではなく、なめらかに曲がった線をパスに追加する、ということもできます。PostScript で扱うことのできるなめらかに曲がった線は、「ベジェ曲線」(Bézier curve) と呼ばれるものです。

ベジェ曲線の形は、「制御点」(control point) と呼ばれる 4 個の点によって指定されます。それらの 4 個の点には順番がありますので、その順番にしたがって、それらを、1、2、3、4、と番号で呼ぶことにしましょう。ベジェ曲線は、まず、制御点 1 からスタートして、1 と 2 とをつなぐ直線に沿って進んでいきます。そして、少しずつ向きを変えていって、3 と 4 とをつなぐ直線に接する形で 4 に到達して終わります。

ベジェ曲線をパスに追加したいときは、curveto というオペレーターを使います。curveto を実行するときには、あらかじめ、6 個の数値をスタックにプッシュしておく必要があります。6 個の数値というのは、プッシュする順番で言うと、制御点 2 の x 座標と y 座標、制御点 3 の x 座標と y 座標、制御点 4 の x 座標と y 座標です。制御点 1 の座標をプッシュする必要がないのは、カレントポイントが制御点 1 になるからです。

curveto を使ってベジェ曲線をパスに追加すると、カレントポイントは、そのベジェ曲線の制御点 4（つまりベジェ曲線の終了点）へ移動します。

プログラムの例 bezier.ps

```
%!PS-Adobe-3.0
newpath
140 400 moveto
100 600 500 700 300 400 curveto
40 setlinewidth 0.5 1 1 setrgbcolor stroke
140 400 moveto
```

```
100 600 lineto
500 700 moveto
300 400 lineto
1 setlinewidth 1 0 0 setrgbcolor stroke
140 100 moveto
100 300 300 100 500 400 curveto
40 setlinewidth 0.5 1 1 setrgbcolor stroke
140 100 moveto
100 300 lineto
300 100 moveto
500 400 lineto
1 setlinewidth 1 0 0 setrgbcolor stroke
showpage
```

4.2 ベジェ曲線の連結

ベジェ曲線とベジェ曲線とを連結することによって、なめらかにつながった 1 本の曲線を描画するためには、ひとつ目のベジェ曲線の制御点 3、2 本のベジェ曲線の連結点、そして二つ目のベジェ曲線の制御点 2、という 3 個の点が、1 本の直線の上に並ぶようにする必要があります。もしもそれらの点が 1 本の直線の上にないとすると、それらの 2 本のベジェ曲線は、それらの連結点のところで折れ曲がることになります。

プログラムの例 bezbez.ps

```
%!PS-Adobe-3.0
newpath
140 600 moveto
100 700 350 700 250 600 curveto
300 500 500 400 400 600 curveto
20 setlinewidth 0.6 1 0.6 setrgbcolor stroke
140 600 moveto
100 700 lineto
350 700 moveto
250 600 lineto
1 setlinewidth 1 0 0 setrgbcolor stroke
250 600 moveto
300 500 lineto
500 400 moveto
400 600 lineto
0 0 1 setrgbcolor stroke
140 300 moveto
100 400 350 400 250 300 curveto
100 150 500 100 400 300 curveto
20 setlinewidth 0.6 1 0.6 setrgbcolor stroke
140 300 moveto
100 400 lineto
350 400 moveto
250 300 lineto
1 setlinewidth 1 0 0 setrgbcolor stroke
250 300 moveto
100 150 lineto
500 100 moveto
400 300 lineto
0 0 1 setrgbcolor stroke
showpage
```

5 塗りつぶし

5.1 塗りつぶしの基礎

直線や曲線によって構成されるグラフィックスではなくて、何らかの領域を塗りつぶすことによってできるグラフィックスを描画したいときは、どうすればいいのでしょうか。

stroke というオペレーターを使うことによってパスを描画することができるわけですが、パスを描画するオペレーターは stroke だけではありません。fill というオペレーターがあって、これもやはりパスを描画します。ただし、fill は、パスを構成する直線や曲線そのものを描画するのではなくて、それらの線によって囲まれた領域を塗りつぶします。

ですから、領域を塗りつぶすことによってできるグラフィックスを描画したいときは、その領域を取り囲む直線や曲線をパスに追加したのちに、stroke の代わりに fill を実行することによってその内部を塗りつぶせばいい、ということになります。

なお、fill が領域を塗りつぶすために使う色は、stroke と同じで、setrgbcolor によって設定された色です。また、パスを描画したのちにそのパスを空にするという点も、fill と stroke とは同じです。

プログラムの例 fill.ps

```
%!PS-Adobe-3.0
0.5 0.8 0 setrgbcolor
newpath
200 100 moveto
200 0 rlineto
0 100 rlineto
-100 100 rlineto
fill
300 500 100 90 360 arc
fill
showpage
```

5.2 グラフィック状態の保存

ところで、ひとつのパスについて、その内部と輪郭の両方を描画したい、というときはどうすればいいのでしょうか。

ひとつのパスに対して fill と stroke の両方を実行すれば、内部と輪郭の両方を描画することができるわけですが、しかしそのためには、どちらか一方を実行した時点でパスの内容が空になってしまうという問題をクリアする必要があります。

ひとつのパスに対して内部と輪郭の両方を描画するためには、パスをどこかに保存してから一方を描画して、そののちパスを回復して、それから残ったほうを描画する、という手順を実行する必要があります。

パスを保存したいときは、gsave というオペレーターを使います。gsave は、パスだけではなく、カレントポイントや色や線の幅などのさまざまな状態から構成される、「グラフィック状態」(graphics state) と呼ばれるものを保存するオペレーターです。

gsave は、grestore というオペレーターとで一組のペアになっています。grestore は、保存されているグラフィック状態を回復する、という動作をするオペレーターです。

ちなみに、グラフィック状態は、「グラフィック状態スタック」(graphics state stack) と呼ばれるスタックに保存されるようになっています。つまり、gsave はグラフィック状態をそのスタックにプッシュするオペレーターで、grestore はポップするオペレーターだということです。

プログラムの例 gsave.ps

```
%!PS-Adobe-3.0
10 setlinewidth
newpath
200 100 moveto
200 0 rlineto
0 100 rlineto
```

```
-100 100 rlineto
0.6 1 0.6 setrgbcolor
gsave fill grestore
0 0 0.8 setrgbcolor
stroke
300 500 100 90 360 arc
0.6 0.6 1 setrgbcolor
gsave fill grestore
0.6 0.4 0 setrgbcolor
stroke
showpage
```

5.3 交差したパスの塗りつぶし

自分自身と交差しているパスを塗りつぶす場合には、パスの内部なのか外部なのかという判断が複雑になります。その判断をするための規則としては、「ワインディング規則」(winding rule) と呼ばれるものと「奇偶規則」(even-odd rule) と呼ばれるものの 2 種類があって、`fill` は前者の規則にしたがってパスの内部かどうかの判断をします。

PostScript には、パスの内部を塗りつぶすオペレーターとして、`fill` のほかにもうひとつ、`eofill` というオペレーターがあります。`eofill` は、パスの内部かどうかの判断に奇偶規則を使います。

プログラムの例 `eofill.ps`

```
%!PS-Adobe-3.0
0 0.5 1 setrgbcolor
newpath
200 300 moveto
100 0 rlineto
0 100 rlineto
-100 0 rlineto
180 320 moveto
100 0 rlineto
0 100 rlineto
-100 0 rlineto
220 340 moveto
100 0 rlineto
0 100 rlineto
-100 0 rlineto
fill
200 100 moveto
100 0 rlineto
0 100 rlineto
-100 0 rlineto
180 120 moveto
100 0 rlineto
0 100 rlineto
-100 0 rlineto
220 140 moveto
100 0 rlineto
0 100 rlineto
-100 0 rlineto
eofill
showpage
```

6 テキスト

6.1 テキストのトークン

文字を並べることによってできるデータのことを「テキスト」(text) と言います。PostScript のプログラムの中に、処理の対象としてテキストを書きたい場合は、

(テキスト)

というように、そのテキストを丸括弧で囲んだものを書きます。たとえば、プログラムの中に、

```
(The Art of PostScript Programming)
```

という記述を書いたとすると、その丸括弧の中のテキストは、処理の対象とみなされることになります。

テキストを構成するそれぞれの文字は、コンピュータの内部では、「文字コード」(character code) と呼ばれる整数によって表現されています。PostScript では、プログラムの中にテキストそのものを書くという方法のほかに、そのテキストを構成するそれぞれの文字に対応する文字コードを並べるという方法でも、テキストをあらわす記述を書くことが可能です。

テキストを文字コードで記述したいときは、

< 16 進数 ··· >

というように、小なりと大なりで囲んだ中に、文字コードをあらわす 16 進数を並べて書きます (小なりと大なりのあいだでは、空白や改行を使って 16 進数を自由に区切ることができます)。たとえば、

```
<49 20 6c 6f 76 65
 20 79 6f 75 2e>
```

という記述は、テキストを 16 進数の列であらわしたものです。

PostScript のプログラムの中に、テキストを丸括弧で囲んだもの、または小なりと大なりで 16 進数の列を囲んだものを書くと、それは、1 個のトークンとして扱われることになります。PostScript のインタプリタがプログラムの中でテキストのトークンに遭遇した場合の処理は、ただ単にそのトークンをスタックにプッシュするだけです。

6.2 テキストの描画

PostScript のインタプリタは、直線や円弧描画するのと同様に、平面の上にグラフィックスとしてテキストを描画することもできます。たとえば、次のプログラムは、 Sakuranomiya というテキストを描画するものです。

プログラムの例 text.ps

```
%!PS-Adobe-3.0
0 0.6 0.4 setrgbcolor
/Times-Roman findfont 64 scalefont setfont
100 600 moveto
(Sakuranomiya) show
showpage
```

このプログラムの中にある show というのが、テキストを描画するオペレーターです。ひとつのテキストをスタックにプッシュしてから show を実行すると、 show は、そのテキストをポップして、それをカレントページに描画します。

show は、カレントポイントを開始点としてテキストを描画していきます。ですから、テキストを描画するときには、 moveto または rmoveto を使って、描画する位置を指定する必要があります。なお、 show は、テキストを描画したのち、そのテキストの末尾にカレントポイントを移動させます。

描画するテキストは、文字コードをあらわす 16 進数の列で書きあらわしてもかまいません。次のプログラムは、文字コードであらわされた Hotarugaike というテキストを描画するものです。

プログラムの例 chacode.ps

```
%!PS-Adobe-3.0
0.4 0 0.6 setrgbcolor
/Times-Roman findfont 64 scalefont setfont
100 600 moveto
<48 6f 74 61 72 75 67 61 69 6b 65> show
showpage
```

6.3 フォント辞書

テキストを描画するためには、テキストを構成するそれぞれの文字をどのような形状で描画するかという情報が必要になります。そのような情報は、「フォント」(font) と呼ばれます。

PostScript では、フォントをあらわしているデータのことを「フォント辞書」(font dictionary) と呼びます。PostScript では、

Times-Roman、 Helvetica、 Courier、 Symbol

というような名前のフォント辞書を使うことができます。

フォント辞書を使うためには、まず最初に、使いたいフォント辞書を探し出すという処理をする必要があります。

フォント辞書は、findfont というオペレーターを使うことによって探し出すことができます。探し出したいフォントの名前をスタックにプッシュしてから findfont を実行すると、findfont は、そのフォントのフォント辞書を探し出して、それをスタックにプッシュします。

「名前」(name) というのも、PostScript のプログラムによって扱われるデータの種類のひとつです。PostScript のインタプリタは、プログラムの中で名前に遭遇した場合、名前そのものを処理するのではなくて、その名前によってあらわされているものを処理します。

それとは逆に、名前によってあらわされているものを処理するのではなくて、名前そのものを処理してほしいときは、トークンとして、名前の左側にスラッシュ(/) を付けたものを書きます。たとえば、プログラムの中に、

/Times-Roman

というトークンを書いたとすると、PostScript のインタプリタは、そのトークンをただ単にスタックにプッシュします。

プログラムの例 font.ps

```
%!PS-Adobe-3.0
0.6 0 0.4 setrgbcolor
/Times-Roman findfont 64 scalefont setfont
100 600 moveto
(Kawaramachi) show
/Helvetica findfont 64 scalefont setfont
100 500 moveto
(Takarazuka) show
/Courier findfont 64 scalefont setfont
100 400 moveto
(Shinkaichi) show
/Symbol findfont 64 scalefont setfont
100 300 moveto
(Kitasenri) show
showpage
```

6.4 日本語のフォント

日本語のテキストを描画するためには、日本語のフォントを使う必要があります。PostScript には、

Ryumin-Light-83pv-RKSJ-H、 GothicBBB-Medium-83pv-RKSJ-H

というような名前を持つ日本語のフォント辞書があります。

なお、プログラムの中に日本語のテキストをあらわす記述を書く場合は、丸括弧の中に日本語の文字をそのまま書くのではなくて、小なりと大なりを使って、文字コードの列を 16 進数で書く必要があります。

プログラムの例 nihongo.ps

```
%!PS-Adobe-3.0
0 0.6 0.8 setrgbcolor
/Ryumin-Light-83pv-RKSJ-H findfont 64 scalefont setfont
100 600 moveto
<90bc 9286 9387 93ec 95fb> show
/GothicBBB-Medium-83pv-RKSJ-H findfont 64 scalefont setfont
100 400 moveto
<91be 8e71 8bb4 8da1 8e73> show
showpage
```

6.5 文字の大きさの変更

探し出されたばかりのフォント辞書は、ひとつひとつの文字の大きさが 1 ポイントだというデータを含んでいます。このデータをそのままにしておくと、1 ポイントの大きさで文字が描画されることになります。それ以外の大きさで文字を描画するためには、フォント辞書の中にある文字の大きさのデータを変更する必要があります。

文字の大きさは、scalefont というオペレーターを実行することによって変更することができます。スタックの先頭に文字の大きさ（単位はポイント）があって、そのひとつ奥にフォント辞書があるときに scalefont を実行すると、scalefont は、スタックからそれらをポップして、文字の大きさのデータが変更されたフォント辞書をスタックにプッシュします。

プログラムの例 scafont.ps

```
%!PS-Adobe-3.0
1 0 0.6 setrgbcolor
/Times-Roman findfont 24 scalefont setfont
100 600 moveto
(Kawanishinoseguchi) show
/Times-Roman findfont 36 scalefont setfont
100 510 moveto
(Kawanishinoseguchi) show
/Times-Roman findfont 50 scalefont setfont
100 400 moveto
(Kawanishinoseguchi) show
showpage
```

6.6 カレントフォント

テキストを描画するためには、かならず、そのために使うフォント辞書をあらかじめ設定しておく必要があります。描画で使うために設定されているフォント辞書は、「カレントフォント」(current font) と呼ばれます。

フォントをカレントフォントとして設定するためには、setfont というオペレーターを実行する必要があります。setfont は、スタックからフォント辞書をポップして、それをカレントフォントとして設定するオペレーターです。

以上のことをまとめると、テキストを描画するためには、そのための準備として次のような作業をする必要がある、ということになります。

(1) findfont を使ってフォント辞書を探し出す。

(2) scalefont を使って文字の大きさを変更する。

(3) setfont を使ってフォント辞書をカレントフォントとして設定する。

6.7 テキストの輪郭

テキストは、 show というオペレーターを使うことによって描画することができるわけですが、テキストを描画する方法はそれだけではありません。テキストの輪郭を構成している直線や曲線をカレントパスに追加して、そののち、 fill や stroke などを使ってカレントパスを描画する、という方法でテキストを描画することも可能です。

テキストの輪郭をカレントパスに追加したいときは、 charpath というオペレーターを使います。1 個のテキストと 1 個の真偽値をスタックにプッシュして、それから charpath を実行すると、 charpath は、それらをポップして、そのテキストの輪郭をカレントパスに追加します。

「真偽値」(boolean) というのは、条件が成り立っているかどうかということを意味するデータのことです。条件が成り立っているというデータを「真」(true) と呼び、条件が成り立っていないというデータを「偽」(false) と呼びます。PostScript では、真を意味する true という名前と、偽を意味する false という名前が、あらかじめ定義されています。

charpath を実行するためには、1 個の真偽値をスタックにプッシュする必要があるわけですが、普通は、偽をスタックにプッシュします。

なぜ偽をプッシュするのかということについて、ごく簡単に説明しておくことにしましょう。フォントは、文字の形をどのように表現するのかということによって、いくつかの種類に分類することができます。それらの種類のうち、現在は、「アウトラインフォント」(outline font) と呼ばれる、直線や曲線を使って文字の輪郭を表現するという形式のフォントが主流になっています。 charpath を実行する前にスタックにプッシュする真偽値というのは、アウトラインフォントではないフォントを使う場合には意味があるのですが、アウトラインフォントを使う場合は、真でも偽でも、どちらも同じ結果になります。ただし、真をプッシュすると余分な処理が実行されますので、アウトラインフォントを使う場合は偽をプッシュほうがいい、ということになるわけです。

プログラムの例 charpath.ps

```
%!PS-Adobe-3.0
newpath
/Times-Roman findfont 140 scalefont setfont
0 0.4 0.8 setrgbcolor
100 500 moveto
(Tenma) false charpath
fill
/Ryumin-Light-83pv-RKSJ-H findfont 200 scalefont setfont
100 250 moveto
<9356 969e> false charpath
0.8 0.4 0 setrgbcolor
fill
showpage
```

テキストの輪郭だけを描画したり、テキストの輪郭と内部とを異なる色で描画したい、という場合には、 show ではなく charpath を使う必要があります。

プログラムの例 outline.ps

```
%!PS-Adobe-3.0
newpath
/Helvetica findfont 120 scalefont setfont
0 0.6 0.4 setrgbcolor
3 setlinewidth
100 500 moveto
(Umeda) false charpath
stroke
/GothicBBB-Medium-83pv-RKSJ-H findfont 200 scalefont setfont
100 250 moveto
<947e 9363> false charpath
gsave
0 0 0.6 setrgbcolor
10 setlinewidth
stroke
grestore
```

```
0.8 1 0.6 setrgbcolor
fill
showpage
```

7 手続き

7.1 名前

PostScript は、ほかの多くのプログラミング言語と同じように、データに名前を付けることができるという機能を持っています。この機能を使ってデータに適切な名前を付けることは、プログラムを人間にとって読みやすいものにする上で、大きな効果があります。

PostScript では、名前は、英字または数字または大半の特殊文字を並べることによって作ります。たとえば、PostScript では、

```
namako   isoginchaku   umiushi843   602kurage   $@_+*-#!
```

というようなものを名前として使うことができます。このように、ほかのプログラミング言語とは違って、名前の先頭の文字は数字でもかまいません。ただし、`4701`、`5.08`、`38e9` などのような、数値だと解釈されるものを名前として使うことはできません。

名前は、ひとつのトークンとして扱われます。PostScript のインタプリタは、プログラムの中で名前に遭遇した場合、その名前がどんなデータに与えられているものなのかということを調べて、そのデータを処理します。

もしも、名前が与えられているデータではなくて名前そのものをデータとして扱ってほしいというときは、その名前の左側にスラッシュ(`/`) を付けます。つまり、

```
/namako   /isoginchaku   /umiushi843   /602kurage   /$@_+*-#!
```

というように書くわけです。PostScript のインタプリタは、スラッシュの付いた名前に遭遇した場合、ただ単にその名前をスタックにプッシュします。

7.2 変数の定義

名前が付けられたデータは、「変数」(variable) と呼ばれます。そして、データに名前を付けることを、変数を「定義する」(define) と言います。

変数を定義したいときは、まずデータに付ける名前をスタックにプッシュして、次にその名前を付けるデータをプッシュして、それから `def` というオペレーターを実行します。たとえば、

```
/namako 3804 def
```

というように、`namako` という名前と 3804 という整数を、この順番でスタックにプッシュしてから `def` を実行すると、`def` は、3804 というデータに `namako` という名前を付けます。ですから、そのあとで、

```
namako ==
```

というプログラムを実行すると、3804 が出力されることになります。

プログラムの例 `def.ps`

```
%!PS-Adobe-3.0
/length 200 def
newpath
100 100 moveto
length 0 rlineto
0 length rlineto
length neg 0 rlineto
closepath
30 setlinewidth 0.2 0.8 0.6 setrgbcolor stroke
showpage
```

7.3 実行可能配列

何個かのトークンを中括弧で囲んだもののことを、「実行可能配列」(executable array) と呼びます。たとえば、

```
{ 41 53 add 27 24 sub mul == }
```

というのは、実行可能配列の一例です。

実行可能配列は、その全体がひとつのトークンになります。PostScript のインタプリタは、プログラムの中で実行可能配列に遭遇した場合、ただ単にそれをスタックにプッシュします。

実行可能配列というのは、その名前のとおり、実行することのできるトークンの列のことです。実行可能配列を実行したいときは、そのためのオペレーターを使います。実行可能配列を実行するオペレーターにはいくつかのものがあって、それらのうちでもっとも単純な動作をするのは、exec というオペレーターです。スタックの先頭に実行可能配列があるときに exec を実行すると、exec は、その実行可能配列をポップして実行します。たとえば、スタックの先頭に、

```
{ 1234 == }
```

という実行可能配列があるとするときに exec を実行すると、exec がその実行可能配列を実行しますので、1234 が出力されることになります。

7.4 手続きの定義

実行可能配列もデータの一種ですから、それに名前を付けるということができます。名前が付けられた実行可能配列というのも変数の一種なのですが、それだけは、「変数」と呼ばずに「手続き」(procedure) と呼ぶのが一般的です。

PostScript のインタプリタに手続きの名前を処理させると、インタプリタは、その名前が与えられている実行可能配列を求めて、それを実行します。たとえば、

```
/nanasen { 7000 == } def
```

というように nanasen という手続きを定義したとすると、nanasen という名前をインタプリタに処理させることによって、7000 を出力させることができます。

このように、手続きというのはオペレーターと同じようなものだと考えることができます。つまり、手続きを定義するというのは、独自の新しいオペレーターを作るということなのです。

プログラムの例 procedu.ps

```
%!PS-Adobe-3.0

/square {
    100 100 moveto
    length 0 rlineto
    0 length rlineto
    length neg 0 rlineto
    closepath
} def

/length 200 def
newpath
square
30 setlinewidth 0.4 0.4 0.8 setrgbcolor stroke
showpage
```

手続きは、自分が実行される前にスタックにプッシュされたデータをポップしてもかまいません。そして、それとは逆に、自分がプッシュしたデータをスタックに残したまま動作を終了してもかまいません。言い換えれば、手続きは、スタックを媒介にしてデータを外から受け取ったり、データを外へ返したりすることができる、ということです。たとえば、

```
/sanbai { 3 mul } def
```

というように sanbai という手続きを定義したとすると、sanbai は、1 個の数値を受け取って、それを 3 倍した結果を返すことになります。ですから、

```
400 sanbai ==
```

というプログラムを実行することによって、1200 が出力されます。

プログラムの例 argume.ps

```
%!PS-Adobe-3.0

/square {
    moveto
    length 0 rlineto
    0 length rlineto
    length neg 0 rlineto
    closepath
} def

/length 200 def
newpath
100 100 square
230 240 square
320 380 square
180 500 square
30 setlinewidth 0 0.6 0.8 setrgbcolor stroke
showpage
```

7.5 辞書

PostScript のインタプリタの内部には、どんな名前がどんなデータに与えられているかということが記載された一覧表のようなデータがあります。そのようなデータは、「辞書」(dictionary) と呼ばれます。

辞書というのは、名前とデータのペアがいくつか集まったものです。そのペアを構成する要素のうち、名前のほうは「キー」(key) と呼ばれ、データのほうは「値」(value) と呼ばれます。

PostScript のインタプリタの中には、普通のスタックのほかに、「辞書スタック」(dictionary stack) と呼ばれるスタックがあって、そこに辞書をプッシュすることができるようになっています（ちなみに、PostScript のインタプリタの中にある普通のスタックは、正式には「オペランドスタック」(operand stack) と呼ばれます)。

PostScript のインタプリタは、キーに対応する値を求めるときに、辞書スタックの中の辞書を、スタックの入口に近いものから順番に調べていって、最初に発見されたキーと値のペアを採用します。ですから、辞書スタックの中にある複数の辞書のそれぞれに、同一のキーを持つペアがある場合は、それらの辞書のうちでもっとも入口に近いところにあるもののペアだけが有効になります。

データに名前を与える `def` というのは、スタックからポップした名前とデータのペア、つまりキーと値のペアを、辞書スタックの先頭にある辞書に追加する、という動作をするオペレーターです。ちなみに、辞書スタックの先頭にある辞書は、「カレント辞書」(current dictionary) と呼ばれます。

なお、辞書スタックには、インタプリタが起動された時点で、三つの辞書がすでに存在しています。それらの辞書は、スタックの入口に近いものから順番に、`userdict`、`globaldict`、`systemdict` という名前で呼ばれます。

7.6 ローカルな変数

PostScript のインタプリタが起動した直後の状態では、`userdict` という辞書がカレント辞書になっています。ですから、その状態のときに `def` を実行すると、キーと値のペアは `userdict` に登録されることになります。

手続きのような、プログラムのどこからでも呼び出すことのできるものを定義する場合は、そのキーと値のペアを `userdict` に登録するのが普通です。しかし、必要になったときに変数を定義して必要ではなくなったときにそれを破棄したいという場合（つまりローカルな変数を定義し

たいという場合）は、userdict ではなく、新しい辞書を作って、それにキーと値のペアを登録する必要があります。

新しい辞書は、dict というオペレーターを実行することによって生成することができます。dict は、スタックから 1 個の整数をポップして、そののち辞書を生成します。その整数は、辞書の容量、つまりそこに登録することのできるキーと値のペアの個数です。たとえば、

```
3 dict
```

というように、3 をプッシュしてから dict を実行したとすると、dict は、キーと値のペアを 3 個まで登録することのできる辞書を生成します。

dict は、自分が生成した辞書を、辞書スタックではなくてオペランドスタックにプッシュします。ですから、dict が生成した辞書を使うためには、それをオペランドスタックからポップして辞書スタックにプッシュする必要があります。

辞書をオペランドスタックから辞書スタックへ移したいときは、begin というオペレーターを実行します。たとえば、

```
3 dict begin
```

というように書くと、dict によって生成された辞書が辞書スタックにプッシュされます。

begin とは逆に、辞書スタックから辞書をポップしたいときは、end というオペレーターを実行します。ただし、end は、ポップした辞書をオペランドスタックに戻すのではなくて、破棄してしまいます。

それでは、PostScript のインタプリタを使って、実際にローカルな変数を作ってみましょう。まず最初に、

```
/momo 8008 def
```

と入力して、momo という変数を定義してください。この時点で、

```
momo ==
```

と入力すると、8008 が出力されるはずです。ちなみに、この変数は userdict に登録されていますので、ローカルではなくてグローバルな変数です。

次に、新しい辞書を作って、そこに変数を登録してみましょう。まず、

```
1 dict begin
```

と入力して、それから、

```
/momo 7117 def
```

と入力してください。この時点で、

```
momo ==
```

と入力すると、7117 が出力されるはずです。ちなみに、この変数はローカルな変数ということになります。

次に、辞書スタックから辞書をポップして、どうなるか試してみましょう。まず、

```
end
```

と入力して、それから、

```
momo ==
```

と入力してください。すると、8008 が出力されるはずです。

プログラムの例 local.ps

```
%!PS-Adobe-3.0

/rect {
    2 dict begin
    /x 300 def
    /y 350 def
    moveto
    x 0 rlineto
    0 y rlineto
```

```
    x neg 0 rlineto
    closepath
    end
} def

/x 100 def
/y 200 def
newpath
x        y         rect
x 20 add y  30 add rect
x 40 add y  60 add rect
x 60 add y  90 add rect
x 80 add y 120 add rect
10 setlinewidth 0.6 0.8 0 setrgbcolor stroke
showpage
```

ところで、スタックの先頭にあるデータに名前を付けたいというときは、いったいどうすればいいと思いますか。

def は、最初にポップしたデータを値とみなして、次にポップしたデータをキーだとみなして、そして変数を定義します。ということは、スタックの先頭に値があるときに、キーをあとからプッシュして def を実行したのでは、値とキーとの順番が逆になってしまいます。ですから、スタックの先頭のデータに名前を付けたいというときは、まず名前をプッシュして、次に exch を実行して、それから def を実行する必要があります。そうすれば、exch によって値とキーとの順番が入れ換わりますので、変数を正しく定義することができるわけです。たとえば、スタックの先頭に何らかのデータがあるとするとき、

```
/kaki exch def
```

を実行したとすると、そのデータに kaki という名前が与えられることになります。

プログラムの例 exch.ps

```
%!PS-Adobe-3.0

/rect {
    2 dict begin
    /height exch def
    /width exch def
    moveto
    width 0 rlineto
    0 height rlineto
    width neg 0 rlineto
    closepath
    end
} def

newpath
100 200 400  50 rect
300 100 100 400 rect
200 400 250 300 rect
150 350 100 100 rect
250 550 250  50 rect
20 setlinewidth 0.4 0.8 0.6 setrgbcolor stroke
showpage
```

8 座標系の変換

8.1 座標系の変換の基礎

カレントパスをカレントページに描画するときに使われる座標系は、初期状態では、原点がカレントページの左下の隅にあって、x 軸が右を向いていて、y 軸が上を向いています。この座標系は、決して固定されたものではなくて、プログラムの中で移動や拡大や回転などの変更を加え

ることができます。そのような変更は、座標系の「変換」(transformation) と呼ばれます。

座標系は、カレントパスや色や線の幅などと同様に、グラフィック状態を構成する要素のひとつです。このことは、gsave を使って現在の座標系を保存しておけば、座標系を変換してグラフィックスを描画したのちに grestore を使って以前の座標系に戻すことができる、ということを意味しています。ですから、gsave と grestore を使うことによって、座標系を変換した影響がほかの部分に波及しないようにすることができます。

8.2 移動

PostScript の座標系は、その原点を好きなだけ移動させることができます。座標系の原点を移動させたいときは、translate というオペレーターを使います。translate を実行するときには、

tx ty translate

というように、あらかじめ 2 個の数値をスタックにプッシュしておく必要があります。*tx* というのが *x* 軸方向の移動距離で、*ty* というのが *y* 軸方向の移動距離です。たとえば、

```
40 20 translate
```

というように translate を実行したとすると、translate は、*x* 軸方向に 40、*y* 軸方向に 20 だけ、座標系の原点を移動させます。

プログラムの例 trans.ps

```
%!PS-Adobe-3.0

/square {
    1 dict begin
    /length 200 def
    newpath
    0 0 moveto
    length 0 rlineto
    0 length rlineto
    length neg 0 rlineto
    closepath
    stroke
    end
} def

/transsquare {
    gsave
    translate
    square
    grestore
} def

10 setlinewidth 1 0.4 0.4 setrgbcolor
  0   0 transsquare
 80 120 transsquare
250 400 transsquare
showpage
```

8.3 拡大

PostScript の座標系は、*x* 軸方向と *y* 軸方向のそれぞれについて、好きなだけ拡大することができます。座標系を拡大したいときは、scale というオペレーターを使います。scale を実行するときには、

sx sy scale

というように、あらかじめ 2 個の数値をスタックにプッシュしておく必要があります。sx というのが x 軸方向の拡大率で、sy というのが y 軸方向の拡大率です。たとえば、

```
1.4 0.8 scale
```

というように scale を実行したとすると、 scale は、x 軸方向に 1.4 倍、y 軸方向に 0.8 倍だけ座標系を拡大します。

プログラムの例 scale.ps

```
%!PS-Adobe-3.0

/transscalearc {
    gsave
    translate
    scale
    0 0 100 0 360 arc
    stroke
    grestore
} def

10 setlinewidth 0 0.6 0.6 setrgbcolor
1   1   200 200 transscalearc
2.2 0.4 300 500 transscalearc
0.6 3   400 400 transscalearc
showpage
```

8.4 回転

PostScript の座標系の座標軸の向きは、原点を中心として好きなだけ回転させることができます。座標系を回転させたいときは、rotate というオペレーターを使います。rotate を実行するときには、あらかじめ 1 個の数値をスタックにプッシュしておく必要があります。その数値は、座標系を回転させる角度になります（単位は度)。回転の方向は、角度がプラスならば反時計回りです。たとえば、

```
30 rotate
```

というように rotate を実行したとすると、 rotate は、原点を中心として反時計回りに 30 度だけ座標系を回転させます。

プログラムの例 rotate.ps

```
%!PS-Adobe-3.0

/transrotatetext {
    gsave
    translate
    rotate
    0 0 moveto
    (Shigisanguchi) show
    grestore
} def

/Times-Roman findfont 64 scalefont setfont
0 0.6 0.4 setrgbcolor
  0 100 100 transrotatetext
 30 140 190 transrotatetext
-30 120 370 transrotatetext
180 460 480 transrotatetext
showpage
```

a b `eq`	a と b とは等しい (equal)
a b `ne`	a と b とは等しくない (not equal)
a b `gt`	a は b よりも大きい (greater than)
a b `lt`	a は b よりも小さい (less than)
a b `ge`	a は b よりも大きいか、またはそれらは等しい (greater equal)
a b `le`	a は b よりも小さいか、またはそれらは等しい (less equal)

表 2: 関係オペレーター

a b `and`	a かつ b（論理積）
a b `or`	a または b（論理和）
a b `xor`	a または b の片方だけ（排他的論理和）
a `not`	a ではない（論理否定）

表 3: 論理オペレーター

9 選択

9.1 関係オペレーター

2 個のデータのあいだに関係が成り立っているかどうかを調べて、その結果をあらわす真偽値を求めるオペレーターのことを、「関係オペレーター」(relational operator) と言います。関係オペレーターを使うことによって、条件が成り立っているかどうかを調べるという動作を記述することができるようになります。

関係オペレーターには、表 2 で示されているような 6 種類のものがあります。たとえば、PostScript のインタプリタに、

```
8 5 gt ==
```

というプログラムを入力したとすると、`true` が出力され、

```
5 8 gt ==
```

というプログラムを入力したとすると、`false` が出力されます。

9.2 論理オペレーター

演算の対象が真偽値で、演算の結果も真偽値であるようなオペレーターのことを、「論理オペレーター」(logical operator) と言います。論理オペレーターを使うことによって、いくつかの条件が組み合わさってできている、複雑な構造の条件を記述することができるようになります。

論理オペレーターには、表 3 で示されているような 4 種類のものがあります。たとえば、PostScript のインタプリタに、

```
5 8 le 7 7 eq and ==
```

というプログラムを入力したとすると、`true` が出力され、

```
5 8 le 7 4 eq and ==
```

というプログラムを入力したとすると、`false` が出力されます。

9.3 条件による動作の選択

PostScript では、`if` というオペレーターを使うことによって、条件が成り立っているときだけ実行可能配列を実行する、ということができます。

if を使って動作を選択的に実行したいときは、まず真偽値をスタックにプッシュして、それから実行可能配列をプッシュして、それから if を実行します。そうすると、 if は、実行可能配列と真偽値をポップして、その真偽値が true だった場合だけ、実行可能配列を実行します。たとえば、PostScript のインタプリタに、

```
4 7 le { 5858 == } if
```

というプログラムを入力したとすると、5858 が出力されますが、

```
7 4 le { 5858 == } if
```

というプログラムを入力したとしても、何も出力されません。

プログラムの例 if.ps

```
%!PS-Adobe-3.0

/rectwitharc {
    4 dict begin
    /height exch def
    /width exch def
    /y exch def
    /x exch def
    x y moveto
    width 0 rlineto
    0 height rlineto
    width neg 0 rlineto
    closepath
    width height eq {
        3 dict begin
        /half width 2 div def
        /centerx x half add def
        /centery y half add def
        width 0 rmoveto
        0 half rmoveto
        centerx centery half 0 360 arc
        end
    } if
    end
} def

newpath
100 100 400 100 rectwitharc
300 250 200 200 rectwitharc
100 250 150 400 rectwitharc
300 500 150 150 rectwitharc
10 setlinewidth 0.4 0.2 0.8 setrgbcolor stroke
showpage
```

動作を選択的に実行するオペレーターとしては、 if のほかにもうひとつ、 ifelse というオペレーターがあります。 ifelse は、条件が成り立っているかどうかということによって、二つの動作のうちのどちらか一方を選択して実行します。

ifelse を使って動作を選択したいときは、まず真偽値をスタックにプッシュして、それから 2 個の実行可能配列をプッシュして、それから ifelse を実行します。そうすると、 ifelse は、2 個の実行可能配列と真偽値をポップして、その真偽値が true だった場合は、先にプッシュされた実行可能配列を実行して、 false だった場合は、あとからプッシュされた実行可能配列を実行します。たとえば、PostScript のインタプリタに、

```
3 3 eq { 7447 == } { 1818 == } ifelse
```

というプログラムを入力したとすると、7447 が出力され、

```
3 5 eq { 7447 == } { 1818 == } ifelse
```

というプログラムを入力したとすると、1818 が出力されます。

プログラムの例 ifelse.ps

```
%!PS-Adobe-3.0

/pairofrect {
    2 dict begin
    /height exch def
    /width exch def
    moveto
    width 0 rlineto
    0 height rlineto
    width neg 0 rlineto
    closepath
    width height gt {
        width 2 div 0 rmoveto
        0 height rlineto
    } {
        0 height 2 div rmoveto
        width 0 rlineto
    } ifelse
    end
} def

newpath
100 100 250 150 pairofrect
400 100 100 450 pairofrect
100 300 150 250 pairofrect
100 600 400 100 pairofrect
20 setlinewidth 1 0.4 0 setrgbcolor stroke
showpage
```

10 繰り返し

10.1 回数の指定による繰り返し

実行可能配列を実行するいくつかのオペレーターの中には、実行可能配列の実行を 1 回だけではなく何回も繰り返すものもあります。そのような繰り返しをするオペレーターのひとつに、repeat というのがあります。

repeat を使って動作を繰り返したいときは、まず、繰り返したい回数をスタックにプッシュして、次に実行可能配列をプッシュして、それから repeat を実行します。たとえば、PostScript のインタプリタに、

```
7 { 4321 == } repeat
```

というプログラムを入力すると、7 個の 4321 が出力されます。

プログラムの例 repeat.ps

```
%!PS-Adobe-3.0
newpath
/y 120 def
12 {
    100 y moveto
    400 0 rlineto
    /y y 50 add def
} repeat
30 setlinewidth 0.4 0.6 0 setrgbcolor stroke
showpage
```

ところで、このプログラムの中では、

```
/y y 50 add def
```

という変数の定義が何回も繰り返されています。このように、すでに辞書に登録されているのと同一のキーに対して def を実行した場合、def は、新しいペアを登録するのではなくて、すでに

登録されているキーに対応する値を変更します。つまり、辞書の中にあるキーに対応する値は、def を使うことによっていくらでも変更することができるわけです。

10.2 数列の生成による繰り返し

実行可能配列の実行を繰り返すオペレーターとしては、repeat のほかに for というのもあります。for は、等差数列のそれぞれの項をスタックにプッシュしてから実行可能配列を実行する、ということを繰り返します。

for を使って実行可能配列の実行を繰り返したいときは、まず 3 個の数値をスタックにプッシュして、次に実行可能配列をプッシュして、それから for を実行します。スタックにプッシュする 3 個の数値というのは、for が発生させる等差数列を決定するためのものです。プッシュする順番で言うと、1 個目が初項、2 個目が公差、3 個目が終了値です。公差がプラスの場合は項が終了値を超えると繰り返しが終了し、公差がマイナスの場合は項が終了値を下回ると繰り返しが終了します。

たとえば、PostScript のインタプリタに、

```
100 15 200 { == } for
```

と入力したとすると、100、115、130、145、160、175、190、と出力されます。同じように、

```
400 -30 200 { == } for
```

と入力した場合は、400、370、340、310、280、250、220、と出力されます。

プログラムの例 for.ps

```
%!PS-Adobe-3.0
newpath
120 50 670 {
    100 exch moveto
    400 0 rlineto
} for
30 setlinewidth 0.8 0 0.4 setrgbcolor stroke
showpage
```

10.3 グラフィックスの列を作る手続き

複雑なグラフィックスを描画するプログラムを書くときは、グラフィックスを規則的に並べて描画する汎用的な手続きを定義しておくと、かなり重宝します。そこで、そのような手続きの例をいくつか紹介することにしましょう。

次のプログラムの中にある horizontalsequence という手続きは、グラフィックスを描画する実行可能配列を受け取って、そのグラフィックスを x 軸の方向に並べて描画します。

プログラムの例 horiseq.ps

```
%!PS-Adobe-3.0

/circle {
    1 dict begin
    /radius exch def
    newpath
    0 0 radius 0 360 arc
    fill
    end
} def

/square {
    1 dict begin
    /length exch def
    newpath
    0 0 moveto
```

```
    length 0 rlineto
    0 length rlineto
    length neg 0 rlineto
    fill
    end
} def

/horizontalsequence {
    5 dict begin
    /proc exch def
    /times exch def
    /step exch def
    /y exch def
    /x exch def
    times {
        gsave
        x y translate
        proc
        grestore
        /x x step add def
    } repeat
    end
} def

0.6 0.8 0 setrgbcolor
100 520 25 16 { 10 circle } horizontalsequence
120 400 60  7 { 40 circle } horizontalsequence
100 260 16 24 { 10 square } horizontalsequence
100 100 80  5 { 70 square } horizontalsequence
showpage
```

次のプログラムの中にある changecolorsequence という手続きは、グラフィックスを描画する実行可能配列を受け取って、色を少しずつ変化させながら、そのグラフィックスを y 軸の方向に並べて描画します。

プログラムの例 chcolor.ps

```
%!PS-Adobe-3.0

/hline {
    newpath
    0 0 moveto
    0 rlineto
    stroke
} def

/changecolorsequence {
    14 dict begin
    /proc exch def
    /blueend exch def
    /greenend exch def
    /redend exch def
    /blue exch def
    /green exch def
    /red exch def
    /times exch def
    /step exch def
    /y exch def
    /x exch def
    /stepblue blueend blue sub times div def
    /stepgreen greenend green sub times div def
    /stepred redend red sub times div def
    times {
        gsave
        x y translate
        red green blue setrgbcolor
```

```
        proc
        grestore
        /blue blue stepblue add def
        /green green stepgreen add def
        /red red stepred add def
        /y y step add def
    } repeat
    end
} def

20 setlinewidth
100 100 27 22 0.6 1 0 0 0.6 1 { 400 hline }
    changecolorsequence
showpage
```

次のプログラムの中にある matrix という手続きは、グラフィックスを描画する実行可能配列を受け取って、そのグラフィックスを x 軸と y 軸の二つの方向に 2 次元的に並べて描画します。

プログラムの例 matrix.ps

```
%!PS-Adobe-3.0

/ellipse {
    4 dict begin
    /theta exch def
    /sy exch def
    /sx exch def
    /radius exch def
    gsave
    theta rotate
    sx sy scale
    newpath
    0 0 radius 0 360 arc
    fill
    grestore
    end
} def

/matrix {
    8 dict begin
    /proc exch def
    /timesy exch def
    /timesx exch def
    /stepy exch def
    /stepx exch def
    /y exch def
    /x exch def
    timesy {
        /xx x def
        timesx {
            gsave
            xx y translate
            proc
            grestore
            /xx xx stepx add def
        } repeat
        /y y stepy add def
    } repeat
    end
} def

0.8 1 0.4 setrgbcolor
0 0 50 38 14 24 { 30 1 0.5 30 ellipse } matrix
0 0 0.6 setrgbcolor
120 300 18 22 20 14 { 10 1 0.4 -60 ellipse } matrix
showpage
```

次のプログラムの中にある circlesequence という手続きは、グラフィックスを描画する実行可能配列を受け取って、そのグラフィックスを円弧の形に並べて描画します。

プログラムの例 circle.ps

```
%!PS-Adobe-3.0

/circle {
    1 dict begin
    /radius exch def
    newpath
    0 0 radius 0 360 arc
    fill
    end
} def

/square {
    1 dict begin
    /length exch def
    newpath
    0 0 moveto
    length 0 rlineto
    0 length rlineto
    length neg 0 rlineto
    fill
    end
} def

/triangle {
    2 dict begin
    /height exch def
    /base exch def
    newpath
    0 0 moveto
    base 0 rlineto
    base 2 div neg height rlineto
    fill
    end
} def

/circlesequence {
    6 dict begin
    /proc exch def
    /times exch def
    /steptheta exch def
    /radius exch def
    /y exch def
    /x exch def
    gsave
    x y translate
    times {
        gsave
        radius 0 translate
        proc
        grestore
        steptheta rotate
    } repeat
    grestore
    end
} def

0 0.8 0.6 setrgbcolor
300 450 160 22 14 { 20 circle } circlesequence
0.6 0.4 1 setrgbcolor
300 200 220 16 10 { 40 square } circlesequence
```

```
1 0.8 0 setrgbcolor
300 450 200 10 32 { 40 24 triangle } circlesequence
showpage
```

次のプログラムの中にある changecolorswirl という手続きは、グラフィックスを描画する実行可能配列を受け取って、色を少しずつ変化させながら、そのグラフィックスを渦巻の形に並べて描画します。

プログラムの例 swirl.ps

```
%!PS-Adobe-3.0

/circle {
    1 dict begin
    /radius exch def
    newpath
    0 0 radius 0 360 arc
    fill
    end
} def

/changecolorswirl {
    16 dict begin
    /proc exch def
    /blueend exch def
    /greenend exch def
    /redend exch def
    /blue exch def
    /green exch def
    /red exch def
    /times exch def
    /steptheta exch def
    /stepradius exch def
    /radius exch def
    /y exch def
    /x exch def
    /stepblue blueend blue sub times div def
    /stepgreen greenend green sub times div def
    /stepred redend red sub times div def
    gsave
    x y translate
    times {
        gsave
        radius 0 translate
        red green blue setrgbcolor
        proc
        grestore
        /blue blue stepblue add def
        /green green stepgreen add def
        /red red stepred add def
        /radius radius stepradius add def
        steptheta rotate
    } repeat
    grestore
    end
} def

280 360 300 -3 22 84 0.6 0 0 1 0.8 1 { 50 circle }
    changecolorswirl
showpage
```