PostScript 実習マニュアル

第四版
2008 年 7 月 17 日（木）
著者——大黒学

目次

第 1 章　PostScript の基礎

1.1　PostScript とは何か

1.1.1　ページ記述言語

この「PostScript 実習マニュアル」という文章は、PostScript というものについて解説することを目的とするチュートリアルです。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。

ページを記述するための言語、つまり、ページの上に印刷することのできるグラフィックスを記述するための言語は、「ページ記述言語」(page description language) と呼ばれます。

ページ記述言語としては、キヤノン株式会社が開発した LIPS、日本電気株式会社 (NEC) が開発した NPDL、セイコーエプソン株式会社が開発した ESC/Page、京セラ株式会社が開発した PRESCRIBE、Hewlett-Packard Company が開発した HP PCL などがあります。そして、PostScript というのもページ記述言語の一種で、この言語を開発したのは、Adobe Systems Incorporated という会社です。

1.1.2　プログラミング言語

コンピュータによって実行される動作を記述した文書は、「プログラム」(program) と呼ばれます。プログラムを書いて、それをコンピュータに与えることによって、コンピュータに可能な範囲内のあらゆる動作をコンピュータに実行させることができます。

プログラムは、通常、日本語や中国語などの自然言語ではなくて、コンピュータの動作を記述するという目的で作られた言語を使って書かれます。そのような、プログラムを書くための専用の言語は、「プログラミング言語」(programming language) と呼ばれます。

プログラミング言語にはさまざまな種類があります。たとえば、Pascal、C、Java、Scheme、ML、Erlang、Smalltalk、Ruby、Prolog などです。

1.1.3　PostScript の特徴

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、単にページ記述言語であるだけではなくて、プログラミング言語でもある、という特徴です。

PostScript では、グラフィックスは、コンピュータによって実行される動作という形で記述されます。ですから、PostScript を使って書かれた文書は、ほかのプログラミング言語で書かれた文書と同じように、「プログラム」と呼ばれます。

ページ記述言語がプログラミング言語としての能力を持つことの利点は、何らかの手順を実行することによって規則的に生成されるグラフィックスを簡単に記述することができる、というところにあります。

1.1.4　インタプリタ

コンピュータが理解することのできるプログラミング言語は、そのコンピュータの「機械語」(machine language) と呼ばれます。

機械語以外のプログラミング言語で書かれたプログラムをコンピュータに実行させるためには、そのプログラムをコンピュータに理解させるためのプログラムが必要になります。そのようなプログラムは、「言語処理系」(language processor) と呼ばれます。

言語処理系は二つの種類に分類されます。それらの種類は、「コンパイラ」(compiler) と「インタプリタ」(interpreter) と呼ばれます。コンパイラというのは、機械語以外の言語で書かれたプログラムを機械語に翻訳するプログラムのことで、インタプリタというのは、機械語以外の言語で書かれたプログラムを実行するプログラムのことです。PostScript のプログラムは、通常、インタプリタによって実行されます。

プリンターのうちには、PostScript のインタプリタを内蔵しているものがあります。そのようなプリンターは、「PostScript プリンター」(PostScript printer) と呼ばれます。PostScript プリンターは、受け取った PostScript のプログラムを実行して、その結果を紙に印刷することができます。

PostScript のインタプリタのひとつで、Ghostscript と呼ばれるものがあります。これは、パソコンの上で動作する PostScript のインタプリタです。これを使うことによって、PostScript のプログラムを実行した結果を、パソコンのモニターで確認したり、PostScript のインタプリタを

内蔵していないプリンターで印刷したりすることができます。

1.2 Ghostscript の使い方

1.2.1 Ghostscript の起動と終了

この節では、Ghostscript の使い方について説明したいと思います。まず最初は起動と終了です。

Ghostscript は、普通、シェル[1]を使って起動します。起動のコマンドは、Windows の場合は `gswin32` で、Linux の場合は `gs` です。

Ghostscript は、起動すると、

```
GS>
```

というプロンプトを出力します。このプロンプトの右側に PostScript のプログラムを入力してエンターキーを押すと、Ghostscript は、そのプログラムを実行して、そののちふたたびプロンプトを出力します。

それでは、Ghostscript を終了させてみましょう。Ghostscript は、`quit` というプログラムを入力することによって終了させることができます。

1.2.2 Ghostscript との対話

Ghostscript は、

(1) プロンプトを出力する。
(2) プログラムを読み込む。
(3) プログラムを実行する。

という動作を延々と繰り返すように作られています。つまり、人間は、PostScript のプログラムを媒介として、Ghostscript と対話をすることになるわけです。

1.2.3 データの出力

それでは、Ghostscript と対話をしてみましょう。まず最初は、データを出力するプログラムを入力してみます。

PostScript では、データを出力するという動作は、

出力したいもの `==`

という形のプログラムによってあらわされます。`==` というのが、「データを出力する」という動作に与えられた名前で、何を出力するのかということは、その左側に書きます。ですから、

```
437 ==
```

というプログラムを Ghostscript に入力すると、

```
GS>437 ==
437
GS>_
```

というように、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。

1.2.4 計算

次に、数値の計算をするプログラムを Ghostscript に入力してみましょう。PostScript では、加減乗除のそれぞれを、次のような名前であらわします。

`add`　加算。
`sub`　減算。
`mul`　乗算。
`div`　除算。

計算をするプログラムは、

数値　数値　計算の名前

[1] Windows の場合はコマンドプロンプト (cmd.exe) など、Linux の場合は bash など。

と書きます。ただし、計算の結果を出力するためには、

数値 数値 計算の名前 ==

というように、最後に == を書く必要があります。たとえば、

```
5 3 add ==
```

プログラムは、「5 と 3 を加算して、その結果を出力する」という動作をあらわすことになります。このプログラムを Ghostscript に入力すると、

```
GS>5 3 add ==
8
GS>_
```

というように、8 という整数が出力されます。

1.2.5 プログラムのファイル

Ghostscript は、ファイルに格納されているプログラムを実行することも可能です。

まず、テキストエディターを使って次のプログラムを入力して、それを sample.ps という名前のファイルに保存してください。

プログラムの例 sample.ps

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

先ほど説明したとおり、このプログラムの 2 行目は、437 を出力するという動作をあらわしています。しかし、1 行目にある、

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

というのは、ここで初めて登場するものです。これは、こ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示している記述です。PostScript のプログラムをファイルに保存する場合には、その先頭の行に、この記述を書くことになっています。

ちなみに、PostScript のプログラムを格納するファイルには、 .ps という拡張子を付けることになっています。

1.2.6 ファイルに格納されたプログラムの実行

ファイルの中に格納されている PostScript のプログラムは、

(パス名) run

というプログラムによって実行することができます。

それでは、先ほど PostScript のプログラムを保存した、sample.ps というファイルがあるディレクトリがカレントディレクトリになっているシェルで Ghostscript を起動して、

```
(sample.ps) run
```

というプログラムを入力してください。そうすると、

```
GS>(sample.ps) run
437
GS>_
```

というように、437 という整数が出力されるはずです。

1.2.7 イメージウィンドウ

Ghostscript は、起動したときに、「イメージウィンドウ」(image window) と呼ばれるウィンドウを開きます。このウィンドウは、PostScript のプログラムを実行することによって生成されたページを表示するためのものです。

先ほど入力したプログラムはページを生成しないものでしたが、今度はページを生成するプログラムを入力してみましょう。Ghostscript に、

```
newpath 100 100 moveto 500 700 lineto stroke showpage
```

というプログラムを入力してください。そうすると、斜めになった一本の直線がイメージウィンドウに表示されるはずです。そしてエンターキーをもう一度押すと、イメージウィンドウの表示が消えて、ふたたびプロンプトが表示されます。

次に、このプログラムをファイルに保存して、それから、そのファイルの内容を Ghostscript に実行させてみましょう。次のプログラムを sample2.ps という名前のファイルに保存して、そののちそれを Ghostscript に実行させてください。

プログラムの例　sample2.ps

```
%!PS-Adobe-3.0
newpath
100 100 moveto
500 700 lineto
stroke
showpage
```

1.2.8　グラフィカルなインターフェース

次に、GSview というプログラムを使って、先ほど入力したプログラムを実行してみましょう。GSview は、Ghostscript をグラフィカルなユーザーインターフェースで扱うことができるようにするプログラムです。

それでは、GSview を起動して、先ほどプログラムを保存した sample2.ps を開いてください。そうすると、GSview のウィンドウに一本の直線が表示されるはずです。

1.3　文法の基礎

1.3.1　ホワイトスペースと区切り文字

PostScript では、空白 (space)、ヌル文字 (null character)、タブ (tab)、改行 (line feed)、復帰 (carriage return)、改ページ (form feed) という文字を、総称して「ホワイトスペース」(whitespace) と呼びます。また、中括弧 ({}) と角括弧 ([]) を、総称して区切り文字 (separator) と呼びます。

1.3.2　トークン

PostScript のプログラムを区切り文字またはホワイトスペースで区切っていったときのそれぞれの部分は、「トークン」(token) と呼ばれます。たとえば、

```
5 3 add ==
```

というプログラムは、5、3、add、== という 4 個のトークンから構成されています。

トークンとトークンとのあいだには、少なくとも 1 個の区切り文字またはホワイトスペースを書く必要があります。ただし、区切り文字は、トークンの区切りということ以外にも意味を持っていますので、それを書くのは、それが必要になる場合だけです。

ホワイトスペースは、トークンとトークンとのあいだであれば、何個でも好きなだけ書くことができます。また、ホワイトスペースをどのように使っても、プログラムの意味は変化しません。ですから、

```
5 3 add ==
```

というプログラムと、

```
5
  3
    add
      ==
```

というプログラムは、まったく同じ動作をすることになります。

1.3.3　注釈

プログラムを書いているとき、それを読む人間（プログラムを書いた人自身もその中に含まれます）に伝えたいことを、そのプログラムの一部分として書いておきたい、ということがしばしばあります。プログラムの中に書かれたそのような文字列は、「注釈」(comment) と呼ばれます。

注釈は、プログラムを処理するプログラムが、「ここからここまでは注釈だ」ということを認識することができるように、注釈を書くための文法にしたがって書く必要があります。

PostScript では、プログラムの中にパーセント (%) を書くと、その直後から最初の改行までが注釈とみなされます。たとえば、

```
5 3 add == % I am a comment.
```

というプログラムの中に書かれている、

```
I am a comment.
```

という文字列は、注釈とみなされることになります。

プログラムを作成したり修正したりしているとき、その一部分を一時的に無効にしたい、ということがしばしばあります。そのような場合は、無効にしたい部分を削除してしまうと、復元するのに手間がかかりますので、削除するのではなくて、注釈にすることによって無効にするという手段が使われるのが普通です。記述の一部分を注釈にすることによって、それを無効にすることを、その部分を「コメントアウトする」(comment out) と言います。

1.3.4 後置記法

PostScript では、「5 と 3 とを加算する」という動作を、

```
5 3 add
```

と書きます。このような、まず最初に動作の対象となるものを書いて、それに対する動作の種類を最後に書く、という書き方は、「後置記法」(postfix notation) または「逆ポーランド記法」(reverse Polish notation) と呼ばれます。

ちなみに、動作の種類を最初に書く、

```
add 5 3
```

というような書き方は、「前置記法」(prefix notation) または「ポーランド記法」(Polish notation) と呼ばれます。そして、動作の種類を中央に書く、

```
5 add 3
```

というような書き方は、「中置記法」(infix notation) と呼ばれます。

1.4 スタック

1.4.1 スタックとは何か

PostScript のインタプリタは、プログラムを実行するために、「スタック」(stack) と呼ばれるものを使います。スタックというのは、その中にデータを一列に並べて格納しておくことのできる容器の一種です。

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。イメージとしては、行き止まりになっている洞窟のようなものを想像していただくといいでしょう。

スタックの中にデータを詰め込むことを、データを「プッシュする」(push) と言います。データをスタックにプッシュすると、すでにスタックに格納されているデータは、その順番を保ったまま、スタックの奥に押し込まれます。

プッシュとは逆に、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。

スタックからのデータの出し入れは、基本的には、後入れ先出し (last in, first out) で先入れ後出し (first in, last out) です。つまり、最後に入れたものが最初に出てきて、最初に入れたものが最後に出てくるということです。

ちなみに、PostScript のインタプリタは、プログラムを実行するために使うスタックだけではなくて、それ以外の目的で使うスタックも持っています。プログラムを実行するためのスタックは、「オペランドスタック」(operand stack) というのが正式な名前です。

1.4.2 後置記法とスタックとの関係

第 1.3 節で説明したように、PostScript のプログラムというのは後置記法で書かれるわけですが、プログラムが後置記法で書かれるということと、プログラムがスタックを使って実行されるということとのあいだには、きわめて密接な関係があります。後置記法で書かれたプログラムは、スタックを使うことによって、自然な形で実行していくことができます。

1.4.3　オブジェクト

PostScript のプログラムによって操作される、コンピュータのメモリーの中にあるデータは、「オブジェクト」(object) と呼ばれます。

オブジェクトは、動作をあらわしているものとそうでないもの、という二つの種類に分類することができます。

動作をあらわしているオブジェクトは、「実行可能オブジェクト」(executable object) と呼ばれます。

それに対して、動作以外のものをあらわしているオブジェクトは、「リテラルオブジェクト」(literal object) と呼ばれます。たとえば、`437` というトークンがあらわしているオブジェクトは、リテラルオブジェクトに分類されます。

1.4.4　実行可能オブジェクト

実行可能オブジェクトは、さらに、PostScript のインタプリタの中に最初から組み込まれているものとそうでないもの、という二つの種類に分類することができます。

PostScript のインタプリタの中に最初から組み込まれている実行可能オブジェクトは、「オペレーター」(operator) とよばれます。第 1.2 節で紹介した、`==` や `add` などは、オペレーターに与えられている名前です。

それに対して、PostScript のインタプリタの中に最初から組み込まれているわけではなくて、プログラムによって作成される実行可能オブジェクトは、「手続き」(procedure) と呼ばれます。

1.4.5　プログラムの実行

PostScript のインタプリタは、PostScript のプログラムを構成しているそれぞれのトークンがあらわしているオブジェクトを、トークンが並んでいるとおりの順番で処理していきます。オブジェクトに対する処理というのは、次のような動作です。

- それがリテラルオブジェクトならば、それをスタックにプッシュする。
- それが実行可能オブジェクトならば、それがあらわしている動作を実行する。

それでは、PostScript のプログラムがスタックを使うことによってどのように実行されるのかということを、

```
437 ==
```

というプログラムを使って説明しましょう。

PostScript のインタプリタは、まず `437` というトークンを処理します。このトークンがあらわしているのはリテラルオブジェクトですので、インタプリタはそれをスタックにプッシュします。

インタプリタは、次に、`==` というトークンを処理します。このトークンは、「スタックから 1 個のオブジェクトをポップして、それを出力する」という動作をするオペレーターに与えられている名前です。オペレーターというのは実行可能オブジェクトですから、インタプリタはそのオペレーターを実行します。`==` は、スタックから 437 というオブジェクトをポップして、それを出力します。

1.4.6　スタックの中にあるオブジェクトの個数

Ghostscript は、スタックに 1 個以上のオブジェクトが格納されているときは、それらのオブジェクトの個数をプロンプトで人間に知らせてくれます。

それでは、試してみましょう。

```
602 118 599 380
```

というプログラムを Ghostscript に入力してください。そうすると、

```
GS>602 118 599 380
GS<4>_
```

というようにプロンプトが変化します。このプロンプトは、4 個のオブジェクトがスタックに格納されているということをあらわしています。ここで `==` を入力すると、

```
GS<4>==
380
GS<3>_
```

というように、最後にプッシュされた整数が出力されて、スタックに格納されているオブジェクトの個数が 1 だけ減少します。

1.4.7 計算の実行

第 1.2 節で紹介した、add、sub、mul、div というのは、加減乗除の計算をするオペレーターに与えられている名前です。

加減乗除のオペレーターは、「スタックから 2 個のオブジェクトをポップして、それらに対して計算をして、その結果をプッシュする」という動作をします。ですから、

```
5 3 add
```

というプログラムを Ghostscript に入力すると、

```
GS>5 3 add
GS<1>_
```

というように、1 個のオブジェクトがスタックに残されます。ここで == を入力すると、

```
GS<1>==
8
GS>_
```

というように、加算の結果が出力されます。

1.4.8 計算の組み合わせ

PostScript では、何らかの計算が実行された場合、その結果はスタックに残されます。このことによって、いくつかの計算を組み合わせたプログラムも、自然な形で実行することができます。

たとえば、「5 と 3 を加算して、その結果と 7 を乗算する」という動作をあらわすプログラムは、

```
5 3 add 7 mul
```

と書くことができます。このプログラムは、次のように実行されます。

(1) 5 をプッシュする。
(2) 3 をプッシュする。
(3) add を実行する（3 をポップして、5 をポップして、5 と 3 を加算して、その結果 (8) をプッシュする）。
(4) 7 をプッシュする。
(5) mul を実行する（7 をポップして、8 をポップして、8 と 7 を乗算して、その結果 (56) をプッシュする）。

同じように、「100 から、5 と 9 を乗算した結果を減算する」という動作をあらわすプログラムは、

```
100 5 9 mul sub
```

と書くことができます。このプログラムは、次のように実行されます。

(1) 100 をプッシュする。
(2) 5 をプッシュする。
(3) 9 をプッシュする。
(4) mul を実行する（9 をポップして、5 をポップして、5 と 9 を乗算して、その結果 (45) をプッシュする）。
(5) sub を実行する（45 をポップして、100 をポップして、100 から 45 を減算して、その結果 (55) をプッシュする）。

1.4.9 オペランドと結果

実行可能オブジェクト（オペレーターまたは手続き）は、最初に 0 個以上のオブジェクトをスタックからポップして、最後に 0 個以上のオブジェクトをスタックにプッシュします。

実行可能オブジェクトがスタックからポップするオブジェクトは、その実行可能オブジェクトの「オペランド」(operand) と呼ばれます。そして、実行可能オブジェクトがスタックにプッシュ

するオブジェクトは、その実行可能オブジェクトの「結果」(result) と呼ばれます。
　通常、実行可能オブジェクトの動作について説明する場合には、それがどのようなオペランドをポップしてどのような結果をプッシュするのかということを、次のような書式で示します。

$operand_1$ $operand_2$ $\cdots$ $operand_n$ `name` $result_1$ $result_2$ $\cdots$ $result_n$

$operand_1$ $operand_2$ $\cdots$ $operand_n$ はオペランドで、$operand_n$ が最初にポップされるオペランドです。`name` は実行可能オブジェクトの名前です。そして、$result_1$ $result_2$ $\cdots$ $result_n$ は結果で、$result_n$ が最後にプッシュされる結果です。
　たとえば、`add` というオペレーターは、

num_1 num_2 `add` sum

という書式によって示されます。
　オペランドまたは結果が 0 個の場合は、実行可能オブジェクトの名前の左側または右側にマイナス (–) を書くことによって、そのことを示します。たとえば、擬似乱数を発生させる `rand` というオペレーターは、0 個のオペランドをポップしますので、

– `rand` int

という書式によって示されます。同じように、`==` は 0 個の結果をプッシュしますので、

any `==` –

という書式によって示されます。
　スタックの行き止まりは、⊢ という記号によって示されます。たとえば、スタック内のすべてのオブジェクトをポップする `clear` というオペレーターは、

⊢ any_1 $\cdots$ any_n `clear` ⊢

という書式によって示されます。

第 2 章　プログラミング言語としての PostScript

2.1　数値

2.1.1　整数と浮動小数点数

　数値 (number) は、PostScript のプログラムが扱うことのできるオブジェクトの一種です。
　数値は、整数 (integer) と浮動小数点数 (floating point number) という二つの種類に分類することができます。
　整数というのは、小数点以下の端数を持たない数値のことです。それに対して、浮動小数点数というのは、1 個の数字列と、その数字列のどこに小数点があるのかという位置を示す整数、という二つの部分から構成されている数値のことです。

2.1.2　整数のトークン

　`0` から `9` までの数字から構成される列は、10 進数でプラスの整数をあらわしているトークンになります。たとえば、`437` は、437 というプラスの整数をあらわしているトークンです。
　数字の列の左端にマイナス (`-`) という文字を書くと、その全体は、マイナスの整数を生成するトークンになります。たとえば、`-56` というトークンは、マイナスの 56 という整数をあらわしています。
　2 進数や 8 進数や 16 進数のような、10 以外の基数で整数をあらわすトークンを書くことも可能です。そのようなトークンは、

基数 `#` 数字の列

と書きます。たとえば、`2#11111111`、`8#377`、`16#ff` は、いずれも、255 という整数をあらわしています。
　ちなみに、10 以外の基数を使ってマイナスの整数をあらわすトークンを書く、ということはできません。

2.1.3　浮動小数点数のトークン

`0.003` とか `41.56` とか `723.0` というような、`0` から `9` までの数字の列の途中に、小数点の位置を示すドット (.) を 1 個だけ書いたものは、10 進数でプラスの浮動小数点数をあらわしているトークンになります。

整数のトークンの場合と同じように、浮動小数点数のトークンの場合も、左端にマイナス (-) を書くと、その全体は、マイナスの浮動小数点数をあらわすトークンになります。たとえば、`-8.317` は、マイナスの 8.317 という浮動小数点数をあらわしているトークンです。

a `e` b という形のトークンを書くことによって、巨大な数値や微小な数値を簡潔に記述することができます。この形のトークンがあらわしているのは、$a \times 10^b$ という浮動小数点数です。たとえば、`-3.71e-24` は、-3.71×10^{-24} という浮動小数点数をあらわしているトークンです。

2.1.4 算術オペレーター

数値に対する計算を実行するオペレーターは、「算術オペレーター」(arithmetic exec) と呼ばれます。

算術オペレーターには、次のようなものがあります。

num$_1$ *num*$_2$ `add` *sum* **オペレーター**

num$_1$ と *num*$_2$ とを加算します。

num$_1$ *num*$_2$ `sub` *difference* **オペレーター**

num$_1$ から *num*$_2$ を減算します。

num$_1$ *num*$_2$ `mul` *mul* **オペレーター**

num$_1$ と *num*$_2$ とを乗算します。

num$_1$ *num*$_2$ `div` *quotient* **オペレーター**

num$_1$ を *num*$_2$ で除算します。

int$_1$ *int*$_2$ `idiv` *quotient* **オペレーター**

整数 *int*$_1$ を整数 *int*$_2$ で除算して、その結果の小数部分を切り捨てた結果を求めます。

int$_1$ *int*$_2$ `mod` *remainder* **オペレーター**

整数 *int*$_1$ を整数 *int*$_2$ で除算したときの余りを求めます。

num$_1$ `abs` *num*$_2$ **オペレーター**

num$_1$ の絶対値を求めます。

num$_1$ `neg` *num*$_2$ **オペレーター**

num$_1$ の符号を反転した結果を求めます。

num$_1$ `ceiling` *num*$_2$ **オペレーター**

num$_1$ 以上の整数のうちで最小のものを求めます。

num$_1$ `floor` *num*$_2$ **オペレーター**

num$_1$ 以下の整数のうちで最大のものを求めます。

num$_1$ `round` *num*$_2$ **オペレーター**

num$_1$ にもっとも近い整数を求めます。

num$_1$ `truncate` *num*$_2$ **オペレーター**

num$_1$ の小数部分を切り捨てます。

`ceiling`、`floor`、`round`、`trancate` のオペランドが浮動小数点数の場合、その結果は、小数部分が 0 の浮動小数点数です。たとえば、3.7 を `ceiling` した結果は 4.0 になります。

num `sqrt` *real* **オペレーター**

num の平方根を求めます。

angle `sin` *real* **オペレーター**

angle のサイン（正弦）を求めます。

angle `cos` *real* **オペレーター**

angle のコサイン（余弦）を求めます。

num den `atan` *angle* **オペレーター**

num/*den* のアークタンジェント（逆正接）、つまり、タンジェント（正接）が *num*/*den* になるような角度を求めます。

`sin` と `cos` のオペランドと、`atan` の結果の単位は、度 (degree) です。

base exponent `exp` *real* **オペレーター**
base^{exponent}（*base* の *exponent* 乗）を求めます。結果は常に浮動小数点数です。

num `ln` *real* **オペレーター**
num の自然対数（$e = 2.718281\cdots$ を底とする *num* の対数）を求めます。

num `log` *real* **オペレーター**
num の常用対数（10 を底とする *num* の対数）を求めます。

− `rand` *int* **オペレーター**
0 から $2^{31} - 1$ までの範囲で、整数の擬似乱数を発生させます。

int `srand` − **オペレーター**
整数 *int* を種として使って、擬似乱数を発生させる乱数ジェネレーターを初期化します。

− `rrand` *int* **オペレーター**
乱数ジェネレーターの現在の状態をあらわす整数を求めます。この整数をオペランドとして `srand` を実行することによって、乱数ジェネレーターの状態を復元することができます。

2.2　名前

2.2.1　名前の基礎

オブジェクトを指示するために使われるトークンは、「名前」(name) と呼ばれます。すでに何度も登場している、 `==` や `add` や `sub` などは、オペレーターを指示している名前です。

名前をオブジェクトに与えることを、名前をオブジェクトに「束縛する」(bind) と言います。

PostScript のインタプリタに名前を処理させると、インタプリタは、その名前がリテラルオブジェクトに束縛されているならばそれをスタックにプッシュして、実行可能オブジェクトに束縛されているならばそれを実行します。

2.2.2　名前の作り方

名前は、英字、数字、特殊文字を並べることによって作ります。たとえば、

```
namako  uso800  300daigen  uni@tako  $827
```

というような名前を作ることができます。

`300daigen` や `$827` のように、先頭の文字は、数字や特殊文字でもかまいません。ただし、`3e8` のように、数値だと解釈されるものは名前ではありません。

次の文字が含まれている名前を作ることはできません。

```
/  (  )  <  >  {  }  [  ]
```

英字の大文字と小文字は区別されますので、`namako`、`Namako`、`NAMAKO` のそれぞれは、異なる別々の名前だとみなされます。

2.2.3　リテラルオブジェクトとしての名前

名前は、それ自体がひとつのリテラルオブジェクトですので、名前をスタックにプッシュする、ということも可能です。

PostScript のインタプリタに名前を処理させると、インタプリタは、名前そのものではなくて、それが束縛されているオブジェクトを処理します。インタプリタに名前そのものを処理させるためには、名前をあらわしているトークンを書く必要があります。

名前をあらわしているトークンは、スラッシュ(`/`) の右側に名前を書くことによって作ります。たとえば、`namako` という名前をあらわしているトークンは、`/namako` と書きます。

```
GS>/namako ==
/namako
```

2.2.4　束縛

名前をオブジェクトに束縛したいときは、`def` というオペレーターを使います。

名前が *name* で、オブジェクトが *any* だとするとき、

name any `def` −

というように def を実行すれば、*name* が *any* に束縛されます。たとえば、

```
/namako 718 def
```

というプログラムを実行することによって、 namako という名前を 718 という整数に束縛することができます。

```
GS>/namako 718 def
GS>namako ==
718
```

def は、名前がすでに何らかのオブジェクトに束縛されている場合、その名前が束縛されているオブジェクトを変更します。

```
GS>/umiushi 100 def
GS>umiushi ==
100
GS>/umiushi 200 def
GS>umiushi ==
200
GS>/umiushi umiushi 40 add def
GS>umiushi ==
240
```

2.3 文字列

2.3.1 文字列の基礎

文字が並んでできている列は、「文字列」(string) と呼ばれます。文字列は、PostScript のプログラムが扱うことのできるオブジェクトの一種です。

文字列を構成している文字の個数は、その文字列の「長さ」(length) と呼ばれます。たとえば、namako という文字列の長さは 6 です。

長さが 0 の文字列は、「空文字列」(empty string) と呼ばれます。

2.3.2 文字列のトークン

文字列をあらわすトークンを作りたいときは、その文字列を丸括弧で囲みます。たとえば、namako という文字列をあらわすトークンは、 (namako) と書きます。

文字列のトークンの中に、ペアになった丸括弧を書いた場合、それらの丸括弧は、文字列のトークンを作るというという意味ではなくて、丸括弧そのものをあらわしていると解釈されます。たとえば、 (a(b)c) は、 a(b)c という文字列をあらわしているトークンです。

左丸括弧の直後に右丸括弧を書いたもの、つまり () は、空文字列をあらわしているトークンです。

2.3.3 文字列の出力

== は、文字列を、それをあらわすトークンの形に変換して出力します。

```
GS>(namako) ==
(namako)
```

オブジェクトを出力するオペレーターとしては、 == のほかに、 = というものもあります。 = は、文字列をそのままの形で出力します。

```
GS>(namako) =
namako
```

2.3.4 エスケープシーケンス

文字列のトークンの中には、数字や英字や特殊文字など、さまざまな文字を書くことができます。しかし、そのままでは文字列のトークンの中に書くことができない文字というのも存在します。たとえば、改行というのは、そのままでは文字列のトークンの中に書くことができない文字のひとつです。また、左右が対応していない丸括弧も、そのままでは文字列のトークンの中に書くことはできません。

文字列のトークンの中に、そのままでは書くことができない文字を書きたいときは、「エスケープシーケンス」(escape sequence) と呼ばれるものを使う必要があります。

エスケープシーケンスというのは、バックスラッシュ(\) という文字[1]で始まる、1 個の文字を意味する文字列のことです。

エスケープシーケンスには、次のようなものがあります。

`\t`　タブ (tab)。
`\n`　改行 (line feed)。
`\r`　復帰 (carriage return)。
`\f`　改ページ (form feed)。
`\b`　バックスペース (backspace)。
`\(`　左丸括弧。
`\)`　右丸括弧。
`\\`　バックスラッシュ(backslash)。
`\`*ddd*　文字コードが 1 桁から 3 桁までの 8 進数 *ddd* であらわされる文字。

それでは、エスケープシーケンスを含んでいる文字列のトークンをいくつか、`=` に出力させてみましょう。

```
GS>(123\n456) =
123
456
GS>(abc\(def) =
abc(def
GS>(\141\142\143) =
abc
```

2.3.5　改行による文字列の分割

文字列をあらわすトークンは、いくつかの行に分割することも可能です。文字列をあらわすトークンを改行で分割したいときは、改行の直前にバックスラッシュを書きます。

```
GS>(abc\
def) =
abcdef
```

2.3.6　文字コードによる文字列のトークン

文字列のトークンを書く方法としては、その文字列を丸括弧で囲むという方法のほかに、文字列を構成している文字の文字コードを記述するという方法もあります。

文字列のトークンを文字コードで記述したいときは、その文字列を構成している文字の文字コードを 16 進数で記述して、それらの 16 進数の列を小なり (`<`) と大なり (`>`) で囲みます。たとえば、`<616263>` は、`abc` という文字列をあらわしているトークンです。

文字コードによる文字列のトークンの中には、16 進数の数字のほかにホワイトスペースを書くこともできて、ホワイトスペースはトークンの意味に影響を与えません。

```
GS><616263> =
abc
GS><6d 6e
6f> =
mno
```

2.3.7　文字列のオペレーター

文字列に関連するオペレーターとしては、次のようなものがあります。

int `string` *string*　　**オペレーター**

ヌル文字 (文字コードが 0 の文字) を並べることによって、長さが *int* の文字列を生成します。

```
GS>5 string ==
(\000\000\000\000\000)
```

string `length` *int*　　**オペレーター**

文字列 *string* の長さを求めます。

[1] 日本語の環境では、バックスラッシュは円マーク (`¥`) で表示されることもあります。

```
GS>(interdisciplinary) length ==
17
```

string index get *int* **オペレーター**

文字列 *string* から *index* 番目の文字を取り出します。結果は、取り出した文字の文字コードです。文字の番号は 0 から始まります。

```
GS>(abcdefg) 2 get ==
99
```

string index int put – **オペレーター**

文字列 *string* の *index* 番目の文字を、文字コードが *int* の文字に置き換えます。

```
GS>/s (abcdefg) def
GS>s 3 42 put
GS>s ==
(abc*efg)
```

string index count getinterval *substring* **オペレーター**

文字列 *string* から、*index* 番目の文字を先頭とする、長さが *count* の部分文字列を取り出します。

```
GS>(abcdefghijk) 3 4 getinterval ==
(defg)
```

$string_1$ *index* $string_2$ putinterval – **オペレーター**

文字列 $string_1$ の *index* 番目の文字を先頭とする部分文字列を $string_2$ に置き換えます。

```
GS>/s (abcdefghijk) def
GS>s 3 (WXYZ) putinterval
GS>s ==
(abcWXYZhijk)
```

string cvi *int* **オペレーター**

文字列 *string* を整数に変換します。

```
GS>(437) cvi ==
437
```

string cvr *real* **オペレーター**

文字列 *string* を浮動小数点数に変換します。

```
GS>(3.14) cvr ==
3.14
```

string cvn *name* **オペレーター**

文字列 *string* を名前に変換します。

```
GS>(namako) cvn ==
/namako
```

any string cvs *substring* **オペレーター**

オブジェクト *any* をあらわす文字列を生成して、それを文字列 *string* に上書きします。結果は、上書きされた *string* の部分文字列です。

```
GS>437 10 string cvs ==
(437)
```

num radix string cvrs *substring* **オペレーター**

radix を基数として、数値 *num* をあらわす文字列を生成して、それを文字列 *string* に上書きします。結果は、上書きされた *string* の部分文字列です。

```
GS>4095 16 10 string cvrs ==
(FFF)
```

2.4　スタックを操作するオペレーター

2.4.1　オブジェクトの番号

この節では、スタック（厳密に言えばオペランドスタック）を操作するオペレーターをいくつか紹介したいと思います。

スタックを操作するオペレーターの動作について説明する場合、しばしば、スタックの中に詰め込まれているオブジェクトを番号で指示することが必要になります。

スタック内のオブジェクトの番号は、出入口にもっとも近いものが 0 で、奥に向かって順番に、1、2、3、と与えられています。

スタックの 0 番目のオブジェクトは、「スタックの先頭のオブジェクト」と呼ばれることもあります。

2.4.2　オブジェクトの出力

$\vdash any_1 \cdots any_n$ pstack $\vdash any_1 \cdots any_n$ **オペレーター**

== を使ってスタックの中のオブジェクトを先頭から順番に出力します。スタックの内容は変化しません。

```
GS>(namako) 537 801
GS<3>pstack
801
537
(namako)
GS<3>
```

$\vdash any_1 \cdots any_n$ stack $\vdash any_1 \cdots any_n$ **オペレーター**

= を使ってスタックの中のオブジェクトを先頭から順番に出力します。スタックの内容は変化しません。

```
GS>(namako) 537 801
GS<3>stack
801
537
namako
GS<3>
```

2.4.3　オブジェクトのポップ

any pop $-$ **オペレーター**

スタックの先頭のオブジェクトをポップして、それを廃棄します。

```
GS>385 273
GS<2>pop
GS<1>pstack
385
```

$\vdash any_1 \cdots any_n$ clear $\vdash$ **オペレーター**

スタックの中にあるすべてのオブジェクトをポップして、それらを廃棄します。

```
GS>518 237 661 493 504 783
GS<6>clear
GS>
```

2.4.4　オブジェクトの個数

$\vdash any_1 \cdots any_n$ count $\vdash any_1 \cdots any_n\ n$ **オペレーター**

スタックの中にあるオブジェクトの個数を数えます。

```
GS>311 427 503 664 272 987
GS<6>count ==
6
GS<6>
```

2.4.5 オブジェクトの複製

any `dup` $any\ any$ **オペレーター**

スタックの先頭のオブジェクトを複製します。

```
GS>398 dup
GS<2>pstack
398
398
```

$any_n \cdots any_0\ n$ `index` $any_n \cdots any_0\ any_n$ **オペレーター**

マイナスではない整数 n をポップしたのち、スタックの n 番目のオブジェクトを複製します。

```
GS>(a) (b) (c) 2 index
GS<4>pstack
(a)
(c)
(b)
(a)
```

$any_{n-1} \cdots any_0\ n$ `copy` $any_{n-1} \cdots any_0\ any_{n-1} \cdots any_0$ **オペレーター**

マイナスではない整数 n をポップしたのち、スタックの先頭から $n-1$ 番目までのオブジェクトを複製します。

```
GS>(a) (b) (c) 3 copy
GS<6>pstack
(c)
(b)
(a)
(c)
(b)
(a)
```

2.4.6 オブジェクトの順番の入れ替え

$any_1\ any_2$ `exch` $any_2\ any_1$ **オペレーター**

スタックの 0 番目のオブジェクトと 1 番目のオブジェクトとを入れ替えます。

```
GS>(a) (b) (c) exch
GS<3>pstack
(b)
(c)
(a)
```

$any_{n-1} \cdots any_0\ n\ j$ `roll` $any_{(j-1) \bmod n} \cdots any_0\ any_{n-1} \cdots any_{j \bmod n}$ **オペレーター**

整数 j と、マイナスではない整数 n をポップしたのち、$any_{n-1} \cdots any_0$ を j 回だけ回転させます。j がプラスの場合は奥から出入口へ向かって回転し、j がマイナスの場合は出入口から奥へ向かって回転します。

```
GS>(a) (b) (c) (d) (e) 4 3 roll
GS<5>pstack
(b)
(e)
(d)
(c)
(a)
GS<5>clear
GS>(a) (b) (c) (d) (e) 4 -3 roll
GS<5>pstack
(d)
(c)
(b)
(e)
(a)
```

2.4.7　マーク

PostScript は、「マーク」(mark) と呼ばれるオブジェクトを扱うことができます。マークをスタックにプッシュすることによって、スタックの中の位置に対して印を付けるということができます。

マークは、何個でもスタックにプッシュすることができます。

− `mark` *mark* **オペレーター**

スタックにマークをプッシュします。

```
GS>mark ==
-mark-
```

mark $any_1 \cdots any_n$ `counttomark` *mark* $any_1 \cdots any_n$ n **オペレーター**

スタックの先頭からマークまでのあいだにあるオブジェクトの個数を数えます。

```
GS>358 mark 364 283 540 881
GS<6>counttomark ==
4
```

mark $any_1 \cdots any_n$ `cleartomark` − **オペレーター**

スタックの先頭からマークまでのあいだにあるオブジェクトとマークをポップして、それらを廃棄します。

```
GS>711 mark 133 975 468 202
GS<6>cleartomark
GS<1>pstack
711
```

2.5　手続き

2.5.1　手続きについての復習

この節では、手続きについて説明したいと思います。

まず最初に、第 1.4 節で手続きについて説明したことを、復習しておくことにしましょう。

PostScript のプログラムによって操作される、コンピュータのメモリーの中にあるデータは、「オブジェクト」(object) と呼ばれます。

オブジェクトは、「実行可能オブジェクト」(executable object) と「リテラルオブジェクト」(literal object) という二つの種類に分類されます。実行可能オブジェクトというのは動作をあらわしているオブジェクトのことで、リテラルオブジェクトというのは動作以外のものをあらわしているオブジェクトのことです。

実行可能オブジェクトは、さらに、「オペレーター」(operator) と「手続き」(procedure) という二つの種類に分類することができます。オペレーターというのは PostScript のインタプリタの中に最初から組み込まれている実行可能オブジェクトのことで、手続きというのはプログラムによって作成される実行可能オブジェクトのことです。

2.5.2　実行可能配列

手続きは、「実行可能配列」(executable array) と呼ばれるプログラムによって作成されます。

実行可能配列というのは、0 個以上のトークンを中括弧 (`{}`) で囲んだもののことです。たとえば、

```
{ 437 == }
```

というのは、実行可能配列の一例です。

PostScript のインタプリタに実行可能配列を入力すると、インタプリタは、それによってあらわされる手続きを作成して、それをスタックにプッシュします。

```
GS>{ 437 == } ==
{437 ==}
```

2.5.3　手続きの実行

手続きというのは実行可能オブジェクトの一種ですので、実行することが可能です。

手続きを実行する方法の一つとして、 exec というオペレータを使うという方法があります。 exec は、スタックから手続きをポップして、それを実行します。

```
GS>{ 437 == } exec
437
GS>{ 5 3 add == } exec
8
```

2.5.4 手続きの定義

手続きというのはオブジェクトの一種ですので、名前を手続きに束縛するということも可能です。名前を手続きに束縛することを、その手続きを「定義する」(define) と言います。そして、手続きを定義するプログラムは、その手続きの「定義」(definition) と呼ばれます。たとえば、

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

というプログラムは、 nanasen という手続きの定義です。

2.5.5 手続きに束縛されている名前

手続きに束縛されている名前をインタプリタに入力すると、インタプリタは、それが束縛されている手続きを実行します。

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

手続きの定義というのはプログラムの一種ですので、それをファイルに格納しておいて、 run を使ってそれを実行する、ということも可能です。

プログラムの例 hello.ps

```
%!PS-Adobe-3.0
/hello { (Hello, world!) = } def
```

実行例

```
GS>(hello.ps) run
GS>hello
Hello, world!
```

2.5.6 手続きのオペランドと結果

手続きも、オペレーターと同じように、スタックからオペランドをポップしたり、スタックに結果をプッシュしたりすることが可能です。たとえば、

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

と定義された sanbai という手続きは、スタックから 1 個の数値をポップして、それを 3 倍した結果をスタックにプッシュします。

```
GS>/sanbai { 3 mul } def
GS>5 sanbai ==
15
GS>7 sanbai ==
21
```

それでは、今度はもう少し実用的な手続き定義してみましょう。

num square *square* **手続き**

数値 *num* の 2 乗を求めます。

プログラムの例 square.ps

```
%!PS-Adobe-3.0
/square { dup mul } def
```

実行例

```
GS>(square.ps) run
GS>7 square ==
49
```

```
GS>0.5 square ==
0.25
```

int `sum` *sum* **手続き**

1 から *int* までの整数の総和を求めます。

プログラムの例　`sum.ps`

```
%!PS-Adobe-3.0
/sum { dup 1 add mul 2 idiv } def
```

実行例

```
GS>(sum.ps) run
GS>10 sum ==
55
```

string `lastchar` *int* **手続き**

文字列 *string* から末尾の文字を取り出します。結果は、取り出した文字の文字コードです。

プログラムの例　`lastchar.ps`

```
%!PS-Adobe-3.0
/lastchar { dup length 1 sub get } def
```

実行例

```
GS>(lastchar.ps) run
GS>(abcdefg) lastchar ==
103
```

2.6　辞書

2.6.1　辞書とは何か

第 2.2 節で説明したように、名前をオブジェクトに与えることを、名前をオブジェクトに「束縛する」(bind) と言います。

名前をオブジェクトに束縛しておくことができるためには、名前とオブジェクトとの対応がどこかに記録されている必要があります。この対応は、「辞書」(dictionary) と呼ばれるオブジェクトに記録されます。

辞書というのは、オブジェクトのペアを何個でも保持することのできるオブジェクトのことです。辞書が保持しているペアの一方は「キー」(key) と呼ばれ、他方は「値」(value) と呼ばれます。通常、キーは名前で、値は任意のオブジェクトです。

2.6.2　辞書スタック

PostScript のインタプリタは、オペランドスタックとは別に、「辞書スタック」(dictionary stack) と呼ばれるスタックを持っています。

辞書スタックは、辞書だけを格納することのできるスタックです。インタプリタは、名前が何に束縛されているかということを調べる場合、辞書スタックに格納されているそれぞれの辞書を、先頭から奥に向かって順番に検索していきます。そして、最初に見付かったペアの値を、名前が束縛されているオブジェクトとして採用します。ですから、辞書スタックに格納されている複数の辞書のそれぞれに、同一のキーを持つペアが保持されている場合は、もっとも出入口に近い位置にある辞書のペアだけが有効になります。

辞書スタックのもっとも奥には、`systemdict` という名前の辞書があります。この辞書には、名前とオペレーターのペアが保持されています。

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。インタプリタの初期状態では、`userdict` という名前の辞書がカレント辞書になっています。

2.6.3　辞書へのペアの登録

名前をオブジェクトに束縛する `def` というオペレーターは、第 2.2 節ですでに登場していますが、もう一度、ここで改めて紹介しておきたいと思います。

key value def – **オペレーター**

key をキー、*value* を値とするペアをカレント辞書に登録します。カレント辞書の中にすでに *key* をキーとするペアが保持されている場合は、そのペアの値を *value* に置き換えます。

2.6.4 一時的な辞書

プログラムの中で使われる名前の個数は、プログラムが大きくなればなるほど増加していきます。ですから、プログラムが大きくなればなるほど、その中で使われている名前が互いに衝突する確率が高くなっていきます。ですから、プログラミング言語の多くは、名前が有効な範囲を限定するという機能を持っています。この機能を使うことによって、プログラムを書く人間は、同じ名前が別の場所でどのように使われているかということに煩わされることなく、自由に名前を使うことができるようになります。

PostScript では、一時的な辞書を使うことによって、名前が有効な範囲を限定することができます。辞書スタックに辞書をプッシュして、その辞書を使って名前をオブジェクトに束縛して、その辞書をポップする、という操作をすれば、その名前は、それがオブジェクトに束縛された時点から辞書がポップされた時点までが有効な範囲ということになります。

一時的な辞書を生成して、それを辞書スタックにプッシュして、辞書スタックからそれをポップする、という操作は、次のオペレーターを使うことによって実行することができます。

int dict *dict* **オペレーター**

マイナスではない整数 *int* を初期の容量とする辞書を生成して、それをオペランドスタックにプッシュします。 辞書の容量 (capacity) というのは、登録することのできるペアの個数のことです。容量を超えてペアを登録しようとすると、容量はその時点で拡張されます。ただし、容量の拡張にはコストが伴いますので、最初から必要な容量を確保しておくことが肝心です。

dict begin – **オペレーター**

辞書 *dict* を辞書スタックにプッシュします。

– end – **オペレーター**

辞書スタックから辞書をポップします。

```
GS>/namako (sea cucumber) def
GS>namako ==
(sea cucumber)
GS>1 dict begin
GS>/namako 4998 def
GS>namako ==
4998
GS>end
GS>namako ==
(sea cucumber)
```

2.6.5 手続きの中の名前

手続きの中で名前を何かに束縛する場合、一時的な辞書を生成して、そこにそれらのペアを登録することによって、その束縛が手続きの外へ影響を及ぼすことを防ぐことができます。

int itos *string* **手続き**

整数 *int* を文字コードとする文字を、その文字だけから構成される文字列に変換します。

プログラムの例 itos.ps

```
%!PS-Adobe-3.0
/itos {
    1 dict begin
    /s 1 string def
    s 0
    3 -1 roll
    put
    s
    end
} def
```

実行例

```
GS>(itos.ps) run
```

```
GS>/s (umiushi) def
GS>97 itos ==
(a)
GS>s ==
(umiushi)
```

2.6.6　オペランドへの名前の束縛

手続きの定義は、そのオペランドを名前で扱うことによって、そうしない場合よりも分かりやすく書くことができます。

def を使って名前をオブジェクトに束縛するためには、まず名前をスタックにプッシュして、それからオブジェクトをプッシュする必要があります。ところが、名前をオペランドに束縛する場合は、名前をあとからプッシュすることになりますので、それらの順番が反対になってしまいます。ですから、この場合は exch を使って、オペランドと名前の順番を逆にする必要があります。たとえば、namako という名前を、スタックの先頭にあるオペランドに束縛するためには、

```
/namako exch def
```

というように、名前をプッシュしたのちに exch を実行する必要があります。

hour *minute*$_1$ hmtom *minute*$_2$ **手続き**

hour 時間 *minute*$_1$ 分という形式であらわされる時間の長さを、何分という形式に変換します。

プログラムの例　hmtom.ps

```
%!PS-Adobe-3.0
/hmtom {
    2 dict begin
    /minute exch def
    /hour exch def
    hour 60 mul minute add
    end
} def
```

実行例

```
GS>(hmtom.ps) run
GS>2 30 hmtom ==
150
```

2.7　条件

2.7.1　真偽値

プログラムは、しばしば、「この数値は 100 よりも大きいかどうか」とか、「この文字列とあの文字列とは等しいかどうか」というような、何かが成り立っているかどうかということについて判断するという必要に迫られます。そのような、成り立っているかどうかという判断の対象となるものは、「条件」(condition) と呼ばれます。

条件に対する判断を実行すると、その結果として、その条件が成り立っているか成り立っていないかということが得られるます。条件が成り立っているということは「真」(true) と呼ばれ、成り立っていないということは「偽」(false) と呼ばれます。たとえば、「この数値は 100 よりも大きい」という条件は、「この数値」が 107 のときは真で、94 のときは偽です。

真と偽は、総称して「真偽値」(Boolean value) と呼ばれます。

真偽値は、PostScript のプログラムが扱うことのできるオブジェクトの一種です。systemdict によって、true という名前が真に、false という名前が偽に束縛されていますので、これらの名前を使うことによって、真偽値をスタックにプッシュすることができます。

```
GS>true ==
true
GS>false ==
false
```

2.7.2 述語

条件に対する判断を実行したいときは、「述語」(predicate) と呼ばれるものを使います。述語というのは、何らかの条件に対する判断を実行して、その結果の真偽値をスタックにプッシュする、という動作をする実行可能オブジェクトのことです。

PostScript のインタプリタは、いくつかの述語をオペレーターとして持っています。それらのオペレーターは、「関係オペレーター」(relational operator) と「論理オペレーター」(logical operator) という二つの種類に分類することができます。そして、関係オペレーターはさらに、「等値オペレーター」(equality operator) と「比較オペレーター」(comparison operator) という二つの種類に分類することができます。

2.7.3 等値オペレーター

二つのオブジェクトが等しいかどうかという条件に対する判断を実行するオペレーターは、「等値オペレーター」(equality operator) と呼ばれます。等値オペレーターは、次の二つです。

any$_1$ *any*$_2$ `eq` *bool* **オペレーター**

等しい (equal)。*any*$_1$ と *any*$_2$ とが等しいならば真、等しくないならば偽。

```
GS>5 5 eq ==
true
GS>5 8 eq ==
false
```

any$_1$ *any*$_2$ `ne` *bool* **オペレーター**

等しくない (not equal)。*any*$_1$ と *any*$_2$ とが等しくないならば真、等しいならば偽。

```
GS>(abc) (abc) ne ==
false
GS>(abc) (abd) ne ==
true
```

2.7.4 比較オペレーター

二つのオブジェクトのあいだの大小関係について判断するオペレーターは、「比較オペレーター」(comparison operator) と呼ばれます。比較オペレーターは、次の四つです。

num$_1$ *num*$_2$ `gt` *bool* **オペレーター**

よりも大きい (greater than)。*num*$_1$ のほうが *num*$_2$ よりも大きいならば真、そうでなければ偽。

num$_1$ *num*$_2$ `lt` *bool* **オペレーター**

よりも小さい (less than)。*num*$_1$ のほうが *num*$_2$ よりも小さいならば真、そうでなければ偽。

num$_1$ *num*$_2$ `ge` *bool* **オペレーター**

よりも大きいかまたは等しい (greater equal)。*num*$_1$ のほうが *num*$_2$ よりも大きいか、またはそれらが等しいならば真、そうでなければ偽。

num$_1$ *num*$_2$ `le` *bool* **オペレーター**

よりも小さいかまたは等しい (less equal)。*num*$_1$ のほうが *num*$_2$ よりも小さいか、またはそれらが等しいならば真、そうでなければ偽。

```
GS>5 3 gt ==
true
GS>5 5 gt ==
false
GS>5 8 gt ==
false
GS>5 3 ge ==
true
GS>5 5 ge ==
true
GS>5 8 ge ==
false
```

比較オペレーターは、数値だけではなくて、文字列に対しても実行することができます。比較オペレーターを文字列に対して実行した場合、それらの文字列は、「辞書式順序」(lexicographical order) と呼ばれる順序にもとづいて大小関係が判断されます。辞書式順序というのは、辞書の見

出しを並べるときに使われる順序のことです。二つの文字列は、それらを辞書式順序で並べたときに前にあるもののほうが後ろにあるものよりも小さい、と判断されます。

```
GS>(abc) (abd) lt ==
true
GS>(abc) (abcd) lt ==
true
```

2.7.5　論理オペレーター

二つの条件の組み合わせになっているような条件について判断したり、真偽値を反転させたりするオペレーターは、「論理オペレーター」(logical operator) と呼ばれます。論理オペレーターは、次の四つです。

bool$_1$ *bool*$_2$ `and` *bool*$_3$ **オペレーター**

論理積 (conjunction)。*bool*$_1$ と *bool*$_2$ の両方が真ならば真、そうでなければ偽。

```
GS>true true and ==
true
GS>true false and ==
false
GS>false true and ==
false
GS>false false and ==
false
```

bool$_1$ *bool*$_2$ `or` *bool*$_3$ **オペレーター**

論理和 (disjunction)。*bool*$_1$ と *bool*$_2$ の両方またはどちらかが真ならば真、両方が偽ならば偽。

```
GS>true true or ==
true
GS>true false or ==
true
GS>false true or ==
true
GS>false false or ==
false
```

bool$_1$ *bool*$_2$ `xor` *bool*$_3$ **オペレーター**

排他的論理和 (exclusive disjunction)。*bool*$_1$ と *bool*$_2$ の一方が真で他方が偽ならば真、両方が同じ真偽値ならば偽。

```
GS>true true xor ==
false
GS>true false xor ==
true
GS>false true xor ==
true
GS>false false xor ==
false
```

bool$_1$ `not` *bool*$_2$ **オペレーター**

否定 (negation)、つまり真偽値の反転。*bool*$_1$ が真ならば偽、偽ならば真。

```
GS>true not ==
false
GS>false not ==
true
```

2.7.6　述語の定義

PostScript のインタプリタの中にはさまざまな述語が組み込まれているわけですが、それらの述語は、きわめて単純な条件についての判断しかできません。しかし、述語というのは、プログラムを書く人が自分で定義することも可能ですので、複雑な条件の判断を記述したいときは、その判断をする述語を自分で定義するといいでしょう。述語を定義するというのは、言い換えれば、結果として真偽値を求める手続きを定義するということです。

それでは、述語を実際に定義してみましょう。

int even *bool* **手続き**

整数 *int* が偶数ならば真、そうでなければ偽。

プログラムの例 even.ps

```
%!PS-Adobe-3.0
/even { 2 mod 0 eq } def
```

実行例

```
GS>(even.ps) run
GS>6 even ==
true
GS>7 even ==
false
```

$string_1$ $string_2$ samelen *bool* **手続き**

文字列 $string_1$ と文字列 $string_2$ が同じ長さならば真、そうでなければ偽。

プログラムの例 samelen.ps

```
%!PS-Adobe-3.0
/samelen {
    2 dict begin
    /string2 exch def
    /string1 exch def
    string1 length string2 length eq
    end
} def
```

実行例

```
GS>(samelen.ps) run
GS>(namako) (hitode) samelen ==
true
GS>(namako) (umiushi) samelen ==
false
```

any_1 any_2 any_3 triple *bool* **手続き**

any_1 と any_2 と any_3 がすべて等しいならば真、そうでなければ偽。

プログラムの例 triple.ps

```
%!PS-Adobe-3.0
/triple {
    3 dict begin
    /any3 exch def
    /any2 exch def
    /any1 exch def
    any1 any2 eq any1 any3 eq and
    end
} def
```

実行例

```
GS>(a) (a) (a) triple ==
true
GS>(a) (a) (b) triple ==
false
```

2.8 選択

2.8.1 選択とは何か

この節では、「選択」(selection) というものについて説明したいと思います。

「選択」という言葉は、「いくつかのものの中からどれかを選び出すこと」というのが一般的な意味ですが、プログラミングに関連する用語としては、「いくつかの動作の中からどれかひと

つを選び出して実行するという動作」という意味で使われます。

2.8.2　選択のオペレーター

PostScript では、選択を実行したいときは、ifelse または if というオペレーターを使います。

bool $proc_1$ $proc_2$ ifelse　−　　**オペレーター**

真偽値 *bool* が真ならば手続き $proc_1$ を実行して、偽ならば手続き $proc_2$ を実行します。

```
GS>true { (It's true.) = } { (It's false.) = } ifelse
It's true.
GS>false { (It's true.) = } { (It's false.) = } ifelse
It's false.
```

bool proc if　−　　**オペレーター**

真偽値 *bool* が真ならば手続き *proc* を実行して、偽ならば何もしません。

```
GS>true { (It's true.) = } if
It's true.
GS>false { (It's true.) = } if
GS>_
```

それでは、選択のオペレーターを使って、手続きを定義してみましょう。

num zero *string*　　**手続き**

数値 *num* がゼロならば zero という文字列をプッシュして、そうでなければ not zero という文字列をプッシュします。

プログラムの例　zero.ps

```
%!PS-Adobe-3.0
/zero { 0 eq { (zero) } { (not zero) } ifelse } def
```

実行例

```
GS>(zero.ps) run
GS>0 zero ==
(zero)
GS>5 zero ==
(not zero)
```

int evenodd *string*　　**手続き**

整数 *int* が偶数ならば even という文字列をプッシュして、奇数ならば odd という文字列をプッシュします。

プログラムの例　evenodd.ps

```
%!PS-Adobe-3.0
/even { 2 mod 0 eq } def
/evenodd { even { (even) } { (odd) } ifelse } def
```

実行例

```
GS>(evenodd.ps) run
GS>6 evenodd ==
(even)
GS>7 evenodd ==
(odd)
```

any pornot　−　　**手続き**

スタックが空ではないならば、その先頭のオブジェクトをポップして出力します。スタックが空ならば何もしません。

プログラムの例　pornot.ps

```
%!PS-Adobe-3.0
/empty { count 0 eq } def
/pornot { empty not { == } if } def
```

実行例

```
GS>(pornot.ps) run
GS>(abc) pornot
(abc)
GS>pornot
GS>_
```

2.8.3 多肢選択

選択の対象となる動作が 3 個以上あるような選択は、「多肢選択」(multibranch selection) と呼ばれます。

多肢選択は、ifelse のオペランドの中に ifelse を実行するプログラムを入れ子にすることによって記述することができます。つまり、

$bool_1$ $proc_1$ {
$bool_2$ $proc_2$ {
⋮
{
$bool_n$ $proc_n$
$proc_{n+1}$ ifelse } ··· } ifelse } ifelse

というように書けばいいわけです。このプログラムは、$bool_1$、$bool_2$、···という真偽値を上から順番にチェックしていって、真を発見したならばその直後にある手続きを実行して、すべてが偽だったならば $proc_{n+1}$ を実行する、という意味です。多肢選択の単純な例をインタプリタに入力すると、次のようになります。

```
GS>false { 1 } {
   false { 2 } {
   true  { 3 }
         { 4 } ifelse } ifelse } ifelse ==
3
```

それでは、多肢選択を実行する手続きを定義してみましょう。

num sign *string* **手続き**

数値 *num* がプラスならば plus という文字列をプッシュして、マイナスならば minus という文字列をプッシュして、ゼロならば zero という文字列をプッシュします。

プログラムの例 sign.ps

```
%!PS-Adobe-3.0
/sign {
    1 dict begin
    /num exch def
    num 0 gt { (plus)  } {
    num 0 lt { (minus) }
             { (zero)  } ifelse } ifelse
    end
} def
```

実行例

```
GS>(sign.ps) run
GS>5 sign ==
(plus)
GS>-5 sign ==
(minus)
GS>0 sign ==
(zero)
```

2.9　繰り返し

2.9.1　繰り返しのオペレーター

この節では、繰り返し (iteration)、つまり何らかの動作を何回も実行するという動作について説明したいと思います。

PostScript では、`repeat`、`for`、`loop`、`forall` などのオペレーターを使うことによって、繰り返しを実行することができます。

2.9.2　回数による繰り返し

決まった回数で動作を繰り返したいときは、`repeat` というオペレーターを使います。

int proc `repeat` −　　**オペレーター**

マイナスではない整数 *int* を回数として手続き *proc* の実行を繰り返します。

```
GS>5 { (namako) = } repeat
namako
namako
namako
namako
namako
```

num int `power` *power*　　**手続き**

数値 *num* と整数 *int* に対して、*num* の *int* 乗 (num^{int}) を求めます。

プログラムの例　`power.ps`

```
%!PS-Adobe-3.0
/power {
    3 dict begin
    /int exch def
    /num exch def
    /p 1 def
    int { /p p num mul def } repeat
    p
    end
} def
```

実行例

```
GS>(power.ps) run
GS>3 4 power ==
81
```

2.9.3　制御変数による繰り返し

`for` というオペレーターは、「制御変数」(control variable) と呼ばれる機構を使うことによって動作を繰り返します。

制御変数は、1 個の数値を保持することができる容器です。制御変数が保持している数値は、制御変数の「値」(value) と呼ばれます。

initial increment limit proc `for` −　　**オペレーター**

initial、*increment*、*limit* は数値で、*proc* は手続きでないといけません。繰り返しに先立って、制御変数の値として *initial* を設定します。そののち、

(1) 終了条件が成り立っているかどうかを判断して、成り立っているならば繰り返しを終了する。
(2) 制御変数の値をスタックにプッシュする。
(3) *proc* を実行する。
(4) 制御変数の値に *increment* を加算した結果を制御変数に設定する。

ということを繰り返します。終了条件は次のとおりです。

increment がプラスの場合　　制御変数の値が *limit* よりも大きい
increment がマイナスの場合　制御変数の値が *limit* よりも小さい

```
GS>1000 30 1100 { == } for
1000
1030
1060
1090
GS>1100 -30 1000 { == } for
1100
1070
1040
1010
```

int alldivisors – **手続き**

整数 *int* のすべての約数を出力します。

プログラムの例 divisor.ps

```
%!PS-Adobe-3.0
/divisor { mod 0 eq } def
/alldivisors {
    2 dict begin
    /int exch def
    1 1 int {
        /i exch def
        int i divisor { i == } if
    } for
    end
} def
```

実行例

```
GS>(divisor.ps) run
GS>7171 alldivisors
1
71
101
7171
```

2.9.4 条件による繰り返し

何らかの条件が成り立つまで動作を繰り返す、というタイプの繰り返しを実行したいときは、通常、loop と exit という二つのオペレーターを組み合わせて使います。

proc loop – **オペレーター**

何らかの手段によって終了させられない限り、手続き *proc* を無限に繰り返します。

– exit – **オペレーター**

repeat、for、loop などの実行を終了させます。

```
GS>(a) (b) (c) { count 0 eq { exit } if == } loop
(c)
(b)
(a)
```

n m gcm *gcm* **手続き**

整数 *n* と整数 *m* の最大公約数 (greatest common measure, GCM) を求めます。

二つの整数の最大公約数は、「ユークリッドの互除法」(Euclidean algorithm) と呼ばれる方法を使うことによって、いとも簡単に求めることができます。ユークリッドの互除法というのは、

ステップ 1　与えられた二つの整数のそれぞれを n と m とする。

ステップ 2　m が 0 ならば計算を終了する。

ステップ 3　n を m で除算したあまりを r とする。

ステップ 4　m を n とする。

ステップ 5　r を m とする。

ステップ 6　ステップ 2 に戻る。

という計算を実行していけば、計算が終了したときの n が、最初に与えられた二つの整数の最大

公約数になっている、というものです。

プログラムの例　gcm.ps

```
%!PS-Adobe-3.0
/gcm {
    3 dict begin
    /m exch def
    /n exch def
    {
        m 0 eq { exit } if
        /r n m mod def
        /n m def
        /m r def
    } loop
    n
    end
} def
```

実行例

```
GS>(gcm.ps) run
GS>54 36 gcm ==
18
```

2.9.5　文字列の要素に対する繰り返し

forall というオペレーターを使うことによって、文字列を構成しているそれぞれの要素（文字コード）に対して何らかの処理を実行する、という動作を繰り返すことができます。

string proc forall　–　　**オペレーター**

文字列 *string* を構成しているそれぞれの要素（文字コード）について、先頭から順番に、それをスタックにプッシュして手続き *proc* を実行する、という動作を繰り返します。

```
GS>(abc) { == } forall
97
98
99
```

s c countc　*n*　　**手続き**

文字列 *s* の中に含まれている、整数 *c* を文字コードとする文字の個数を求めます。

プログラムの例　countc.ps

```
%!PS-Adobe-3.0
/countc {
    3 dict begin
    /c exch def
    /s exch def
    /n 0 def
    s { c eq { /n n 1 add def } if } forall
    n
    end
} def
```

実行例

```
GS>(countc.ps) run
GS>(namamuginamagomenamatamago) (m) 0 get countc ==
6
```

2.10　再帰

2.10.1　再帰とは何か

この節では、「再帰」(recursion) というものについて説明したいと思います。

再帰というのは、全体の一部分として全体と同じ構造のものが含まれているとか、何かを定義するために、定義される当のものが使われている、というような性質のことです。再帰という性

質を持っているものは、「再帰的な」(recursive) と形容されます。

全体の一部分として全体と同じ構造のものが含まれているという構造は、「再帰的な構造」(recursive structure) と呼ばれます。

雑誌の表紙に、ときおり、同じ雑誌の同じ号を手に持った人物が登場することがあります。その人物が持っている雑誌の表紙には、同じ人物がいて、同じ雑誌を持っています。すると、表紙の中に同じ表紙がある、という構造が無限に続くことになります。このような構造は、再帰的な構造の一例です。

何かを定義するために、定義される当のものが使われているという定義は、「再帰的な定義」(recursive definition) と呼ばれます。

たとえば、「子孫」という言葉は、次のように定義することができます。

- B が A の子供であるならば、B は A の子孫である。
- C が A の子供であって、かつ B が C の子孫であるならば、B は A の子孫である。

この定義は、「子孫」という言葉を使って「子孫」という言葉を定義していますので、再帰的な定義だということになります。

2.10.2 基底

再帰的な定義は、選択が可能な二つ以上の記述から構成されていて、それらの選択肢のうちに少なくとも一つ、定義される当のものを使っていないものがある必要があります。なぜなら、すべての選択肢が、定義される当のものを使っているとすると、その定義は堂々巡りになってしまうからです。

再帰的な定義を構成している、選択が可能な記述のうちで、定義される当のものを使っていないものは、その定義の「基底」(basis) と呼ばれます。たとえば、「子孫」という言葉の再帰的な定義は二つの選択肢から構成されているわけですが、「B が A の子供であるならば、B は A の子孫である」という選択肢が、この定義の基底です。

2.10.3 手続きの再帰的な定義

手続きは、再帰的に定義することが可能です。手続きを再帰的に定義するというのは、自分自身という手続きを使って手続きを定義するということです。再帰的な構造を持っている概念を取り扱う手続きは、再帰的に定義するほうが、再帰的ではない方法で定義するよりもすっきりした記述になります。

手続きを再帰的に定義する場合も、それが堂々巡りになることを防ぐために、基底となる選択肢を記述することが必要になります。

2.10.4 階乗

n がマイナスではない整数だとするとき、n から 1 までの整数をすべて乗算した結果、つまり、

$$n \times (n-1) \times (n-2) \times \cdots \times 1$$

という計算の結果のことを、n の「階乗」(factorial) と呼んで、$n!$ と書きあらわします。ただし、$0!$ は 1 だと定義します。

たとえば、$5!$ は、

$$5 \times 4 \times 3 \times 2 \times 1$$

という計算をすればいいわけですから、120 ということになります。

階乗というのは、再帰的な構造を持つ概念の一例です。なぜなら、$n!$ を求める計算は、

$$\begin{cases} & 0! = 1 \\ n \geq 1 \text{ ならば} & n! = n \times (n-1)! \end{cases}$$

ということだと考えることができるからです。

再帰的な構造を持っている概念を取り扱う手続きは、再帰的に定義すると、すっきりした記述になります。ですから、階乗を求める手続きは、自分自身を使うことによって、かなりすっきりと定義することができます。

n `fact` *fact* **手続き**

マイナスではない整数 n の階乗を求めます。

プログラムの例 `fact.ps`

```
%!PS-Adobe-3.0
/fact {
    1 dict begin
    /n exch def
    n 0 eq { 1 } {
    n 1 ge { n n 1 sub fact mul }
           { (undefined) } ifelse } ifelse
    end
} def
```

実行例

```
GS>(fact.ps) run
GS>5 fact ==
120
```

2.10.5　フィボナッチ数列

フィボナッチ数列 (Fibonacci sequence) と呼ばれる数列の第 n 項を求める計算も、再帰的な構造を持っています。

フィボナッチ数列の第 n 項 (F_n) は、

$$\begin{cases} & F_0 = 0 \\ & F_1 = 1 \\ n \geq 2 \text{ ならば} & F_n = F_{n-2} + F_{n-1} \end{cases}$$

という計算によって求めることができます。たとえば、第 7 項を求めたいときは、第 5 項と第 6 項とを加算すればいいわけです。

n `fibona` *fibona* **手続き**

マイナスではない整数 n について、フィボナッチ数列の第 n 項を求めます。

プログラムの例　`fibona.rb`

```
%!PS-Adobe-3.0
/fibona {
    1 dict begin
    /n exch def
    n 0 eq { 0 } {
    n 1 eq { 1 } {
    n 2 ge { n 2 sub fibona n 1 sub fibona add }
           { (undefined) } ifelse } ifelse } ifelse
    end
} def
```

実行例

```
GS>(fibona.ps) run
GS>20 fibona ==
6765
```

2.11　高階手続き

2.11.1　高階手続きとは何か

この節では、「高階手続き」(higher-order procedure) というものについて説明したいと思います。

高階手続きというのは、手続きを取り扱う手続きのことです。手続きは、特殊な動作をするように定義するよりも、スタックからポップした手続きによって動作が定まるように一般的に定義するほうが、応用することのできる範囲が広くなります。

ちなみに、オペレーターの中にも手続きを取り扱うものがあります。たとえば、

`exec`　`ifelse`　`if`　`repeat`　`for`　`loop`　`forall`

などがそうです。このようなオペレーターは、「高階オペレーター」(higher-order operator) と

呼ばれます。

2.11.2 手続きの実行

高階手続きを定義する場合には、ほとんどかならず、スタックからポップした手続きを実行することが必要になります。

スタックからポップした手続きを実行したいときは、exec や ifelse などの高階オペレーターを使うか、または、名前をその手続きに束縛して、その名前を使って手続きを実行します。

それでは、スタックからポップした手続きを実行する、簡単な高階手続きを定義してみましょう。

proc twice – **手続き**

手続き *proc* の実行を二回だけ繰り返します。

プログラムの例 twice.ps

```
%!PS-Adobe-3.0
/twice {
    1 dict begin
    /proc exch def
    proc proc
    end
} def
```

実行例

```
GS>(twice.ps) run
GS>{ (tanpopo) == } twice
(tanpopo)
(tanpopo)
GS>5 40 300 { add } twice ==
345
```

2.11.3 辞書スタックの検索

手続きに束縛されている名前を PostScript のインタプリタに処理させると、インタプリタは、その手続きを実行することになります。しかし、高階手続きを定義するときには、手続きを実行するのではなくて、それをスタックにプッシュすることが必要になる場合もあります。

名前が束縛されている手続きを、その名前を使ってスタックにプッシュしたいときは、辞書スタックを検索することによって、その名前をキーとするペアの値を求める必要があります。辞書スタックは、load というオペレーターを使うことによって検索することができます。

key load *value* **オペレーター**

辞書スタックに格納されている辞書を先頭から順番に検索して、*key* をキーとするペアが発見されたならば、その値をオペランドスタックにプッシュします。

```
GS>/square { dup mul } def
GS>/square load ==
{dup mul}
```

2.11.4 数列の総和

数列を構成しているすべての項を加算した結果は、「数列の総和」(summation of sequence) と呼ばれます。

数列の総和を求める方法としては、最初から具体的な数列の総和を求める手続きを定義してしまってもいいわけですが、数列の項を求める手続きをスタックからポップして総和を求める手続きを定義するほうが、それを使ってさまざまな数列の総和を求めることができますので、汎用性が高くなります。

n f sum *sum* **手続き**

n が 1 以上の整数で、*f* が 1 個の数値をポップして 1 個の数値をプッシュする手続きだとするとき、1、2、・・・、*n* のそれぞれを *f* で処理した結果から構成される数列の総和を求めます。数学の式で書けば、

$$\sum_{i=1}^{n} f(i) = f(1) + f(2) + \cdots + f(n)$$

を求めるということです。たとえば、次のような計算をすることができます。

プログラム	式	結果
`10 { } sum`	$\sum_{i=1}^{10} i = 1 + 2 + \cdots + 10$	55
`10 { dup mul } sum`	$\sum_{i=1}^{10} i^2 = 1^2 + 2^2 + \cdots + 10^2$	385
`10 { dup dup mul mul } sum`	$\sum_{i=1}^{10} i^3 = 1^3 + 2^3 + \cdots + 10^3$	3025

プログラムの例　sum.ps

```
%!PS-Adobe-3.0
/sum {
    2 dict begin
    /f exch def
    /n exch def
    n 1 eq { 1 f } {
    n 2 ge { n f n 1 sub /f load sum add }
           { (undefined) } ifelse } ifelse
    end
} def
```

実行例

```
GS>(sum.ps) run
GS>10 { } sum ==
55
GS>10 { dup mul } sum ==
385
GS>10 { dup dup mul mul } sum ==
3025
```

2.11.5　文字列に対する写像

列を構成しているそれぞれの要素に対して何らかの処理を実行して、それらの結果から構成される列を求める、という処理（またはその処理の結果）は、列に対する「写像」(map) と呼ばれます。

写像を実行する手続きについても、手続きをスタックからポップして、その手続きで列の要素を処理する高階手続きを定義しておけば、さまざまな写像にそれを応用することができます。

s_1 f `maps` s_2　　**手続き**

s_1 が文字列で、f が 1 個の整数（文字コード）をポップして 1 個の整数（文字コード）をプッシュする手続きだとするとき、s_1 を構成しているそれぞれの要素に対して f を実行した結果から構成される文字列を求めます。

プログラムの例　maps.ps

```
%!PS-Adobe-3.0
/maps {
    6 dict begin
    /f exch def
    /s1 exch def
    /n s1 length def
    /s2 n string def
    0 1 n 1 sub {
        /i exch def
        /e s1 i get def
        s2 i e f put
    } for
    s2
    end
} def
```

実行例

```
GS>(maps.ps) run
GS>(tsuyukusa) { 32 sub } maps ==
(TSUYUKUSA)
GS>(password) { pop 42 } maps ==
(********)
```

2.12 配列

2.12.1 配列の基礎

PostScript のプログラムは、「配列」(array) と呼ばれるオブジェクトを扱うことができます。

配列の内部には、何個でも好きなだけ、オブジェクトを一列に並べて格納しておくことができます。配列の全体はひとつのオブジェクトですので、1 回のプッシュやポップで、それをスタックに入れたりスタックから出したりすることができます。

配列を構成しているそれぞれのオブジェクトは、その配列の「要素」(element) と呼ばれます。また、配列を構成している要素の個数は、その配列の「長さ」(length) と呼ばれます。また、長さが 0 の配列は、「空配列」(empty array) と呼ばれます。

2.12.2 配列の生成

要素を列挙することによって配列を生成したいときは、左角括弧 (`[`) と右角括弧 (`]`) というオペレーターを使います。

– `[` *mark* **オペレーター**

スタックにマークをプッシュします。第 2.4 節で紹介した `mark` と同じ動作をするオペレーターです。

mark $any_1 \cdots any_n$ `]` *array* **オペレーター**

スタックの先頭からマークまでのあいだにあるオブジェクトから構成される配列を生成します。それぞれの要素は、$any_1 \cdots any_n$ という順番で並びます。

```
GS>[ 367 28.4 (nazuna) true ] ==
[367 28.4 (nazuna) true]
GS>[ 5 3 add 10 mul 70 33 11 idiv sub ] ==
[80 67]
GS>[ 21 [ 53 74 ] [ 60 [ 53 48 ] 99 ] 18 ] ==
[21 [53 74] [60 [53 48] 99] 18]
GS>[ ] ==
[]
```

2.12.3 配列のオペレーター

配列を扱うオペレーターとしては、次のようなものがあります。

int `array` *array* **オペレーター**

マイナスではない整数 *int* を長さとする配列を生成します。生成される配列は、「ヌルオブジェクト」(null object) と呼ばれるオブジェクトから構成されます。

```
GS>5 array ==
[null null null null null]
```

array `length` *int* **オペレーター**

配列 *array* の長さを求めます。

```
GS>[ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 ] length ==
17
```

array index `get` *int* **オペレーター**

配列 *array* から *index* 番目の要素を取り出します。要素の番号は 0 から始まります。

```
GS>[ (a) (b) (c) (d) (e) ] 2 get ==
(c)
```

array index any `put` – **オペレーター**

配列 *array* の *index* 番目の要素を *any* に置き換えます。

```
GS>/a [ (a) (b) (c) (d) (e) ] def
GS>a 3 (*) put
GS>a ==
[(a) (b) (c) (*) (e)]
```

array index count getinterval *subarray* **オペレーター**

配列 *array* から、*index* 番目の要素を先頭とする、長さが *count* の部分配列を取り出します。

```
GS>[ (a) (b) (c) (d) (e) (f) (g) ] 2 4 getinterval ==
[(c) (d) (e) (f)]
```

$array_1$ *index* $array_2$ putinterval − **オペレーター**

配列 $array_1$ の *index* 番目の要素を先頭とする部分配列を $array_2$ に置き換えます。

```
GS>/a [ (a) (b) (c) (d) (e) (f) (g) ] def
GS>a 2 [ (X) (Y) (Z) ] putinterval
GS>a ==
[(a) (b) (X) (Y) (Z) (f) (g)]
```

$any_0 \cdots any_{n-1}$ *array* astore *array* **オペレーター**

まず最初に配列 *array* をポップします。次に、*array* の長さ（要素の個数）と同じ個数のオブジェクトをポップして、それらのオブジェクトを *array* に格納します（any_i が i 番目の要素になります)。そして最後に *array* をプッシュします。

```
GS>(a) (b) (c) 3 array astore ==
[(a) (b) (c)]
```

array aload $any_0 \cdots any_{n-1}$ *array* **オペレーター**

まず最初に配列 *array* をポップします。次に、*array* のそれぞれの要素を、先頭から順番にプッシュします。そして最後に *array* をプッシュします。

```
GS>[ (a) (b) (c) ] aload
GS<4>pstack
[(a) (b) (c)]
(c)
(b)
(a)
```

array proc forall − **オペレーター**

配列 *array* を構成しているそれぞれの要素について、先頭から順番に、それをスタックにプッシュして手続き *proc* を実行する、という動作を繰り返します。

```
GS>[ (a) (b) (c) ] { == } forall
(a)
(b)
(c)
```

2.12.4　配列を扱う手続き

それでは、配列を扱う手続きを定義してみましょう。

a suma *sum* **手続き**

a が数値を要素とする配列だとするとき、それらの要素をすべて加算した結果を求めます。

プログラムの例　suma.ps

```
%!PS-Adobe-3.0
/suma {
    2 dict begin
    /a exch def
    /sum 0 def
    a { /sum exch sum add def } forall
    sum
    end
} def
```

実行例

```
GS>(suma.ps) run
GS>[ 8000 700 60 5 ] suma ==
8765
```

a_1 reverse a_2 **手続き**

配列 a_1 に対して、それとは逆の順番で要素が並んでいる配列を求めます。

プログラムの例 reverse.ps

```
%!PS-Adobe-3.0
/reverse {
    5 dict begin
    /a1 exch def
    /n a1 length def
    /a2 n array def
    0 1 n 1 sub {
        /i exch def
        /e a1 i get def
        a2 n i sub 1 sub e put
    } for
    a2
    end
} def
```

実行例

```
GS>(reverse.ps) run
GS>[ (a) (b) (c) (d) (e) ] reverse ==
[(e) (d) (c) (b) (a)]
```

n f arrayf $array$ **手続き**

n がマイナスではない整数で、f が 1 個の整数をポップして 1 個のオブジェクトをプッシュする手続きだとするとき、0、1、2、・・・、$n-1$ のそれぞれをプッシュして f を実行した結果から構成される配列を生成します。

プログラムの例 arrayf.ps

```
%!PS-Adobe-3.0
/arrayf {
    4 dict begin
    /f exch def
    /n exch def
    /a n array def
    0 1 n 1 sub {
        /i exch def
        a i i f put
    } for
    a
    end
} def
```

実行例

```
GS>(arrayf.ps) run
GS>16 { dup mul } arrayf ==
[0 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225]
GS>10 { 100 exch sub } arrayf ==
[100 99 98 97 96 95 94 93 92 91]
GS>8 { 2 mod 0 eq } arrayf ==
[true false true false true false true false]
```

a_1 f mapa a_2 **手続き**

a_1 が配列で、f が 1 個のオブジェクトをポップして 1 個のオブジェクトをプッシュする手続きだとするとき、a_1 を構成しているそれぞれの要素に対して f を実行した結果から構成される配列を求めます。

プログラムの例　mapa.ps

```
%!PS-Adobe-3.0
/mapa {
    6 dict begin
    /f exch def
    /a1 exch def
    /n a1 length def
    /a2 n array def
    0 1 n 1 sub {
        /i exch def
        /e a1 i get def
        a2 i e f put
    } for
    a2
    end
} def
```

実行例

```
GS>(mapa.ps) run
GS>[ 38 27 64 55 43 91 ] { 10 mul } mapa ==
[380 270 640 550 430 910]
GS>[ true false false true ] { { 1 } { 0 } ifelse } mapa ==
[1 0 0 1]
GS>[ (sakura) (ume) (mokuren) (momo) ] { length } mapa ==
[6 3 7 4]
```

2.13　ファイル

2.13.1　ファイルオブジェクト

この節では、ファイルに対する操作について説明したいと思います。

ファイルに対して読み込みや書き込みなどの操作を実行するためには、「ファイルオブジェクト」(file object) と呼ばれるオブジェクトを生成する必要があります。ファイルオブジェクトというのは、ファイルに関連付けられたオブジェクトのことです。ファイルに対する操作を実行するオペレーターの多くは、ファイルオブジェクトをスタックからポップして、それを使ってファイルにアクセスします。

2.13.2　ファイルのオープン

ファイルオブジェクトを生成して、それをファイルに関連付けることを、ファイルを「オープンする」(open) と言います。ファイルをオープンしたいときは、`file` というオペレーターを使います。

pathname access `file` *file*　　**オペレーター**

文字列 *access* によって指定された方法でファイルにアクセスするためのファイルオブジェクトを生成して、それを、文字列 *pathname* をパス名とするファイルに関連付けます。

ファイルに対するアクセスの方法を指定するための文字列は、「アクセス文字列」(access string) と呼ばれます。アクセス文字列としては、次のようなものがあります。

`r`　読み込むだけ。対象となるファイルは、すでに存在している必要があります。

`w`　書き込むだけ。対象となるファイルが存在していない場合は、空のファイルを生成します。すでに存在している場合は、書き込みに先立って以前の内容を消去します。

`a`　書き込むだけ。対象となるファイルが存在していない場合は、空のファイルを生成します。すでに存在している場合は、以前の内容の末尾に、新しく書き込んだデータを追加します。

`r+`　読み込みと書き込み。対象となるファイルは、すでに存在している必要があります。

`w+`　読み込みと書き込み。対象となるファイルが存在していない場合は、空のファイルを生成します。すでに存在している場合は、書き込みに先立って以前の内容を消去します。

`a+`　読み込みと書き込み。対象となるファイルが存在していない場合は、空のファイルを生成します。すでに存在している場合は、以前の内容の末尾に、新しく書き込んだデータを追加します。

たとえば、

```
/in (asagao.txt) (r) file def
```

というプログラムを実行することによって、asagao.txt というパス名で指定されるファイルからデータを読み込むためのファイルオブジェクトを生成して、in という名前をそのファイルオブジェクトに束縛することができます。

2.13.3 読み書き位置

ファイルに対する読み書きは、「読み書き位置」(read/write position) と呼ばれる位置に対して実行されます。

r または r+ でファイルをオープンした場合、オープンした直後の読み書き位置は、ファイルの先頭です。

w または w+ でファイルをオープンした場合も、ファイルは空になっているわけですから、最初の読み書き位置はファイルの先頭です。それに対して、a a+ でファイルをオープンした場合、最初の読み書き位置はファイルの末尾になります。

ファイルに対してデータの読み書きを実行するオペレーターは、読み書き位置に対して読み書きを実行して、そののち、次に読み書きを実行することになる位置へ読み書き位置を移動させます。

ファイルの末尾という位置は、「ファイルの終わり」(end of file) と呼ばれます。

2.13.4 ファイルのクローズ

ファイルに対する一連の処理が終了したときは、かならず、ファイルとファイルオブジェクトとの関連を消滅させるという操作をする必要があります。この操作をすることを、ファイルを「クローズする」(close) と言います。

ファイルに対してデータの読み込みを実行するオペレーターは、読み込む前からすでに読み書き位置がファイルの終わりに到達している場合、または読み込んでいる途中で読み書き位置がファイルの終わりに到達した場合、そのファイルをクローズします。

ファイルを明示的にクローズしたいときは、closefile というオペレーターを使います。

file `closefile` – **オペレーター**

ファイルオブジェクト *file* に関連付けられているファイルをクローズします。

2.13.5 読み書きのオペレーター

ファイルに対する読み書きを実行するオペレーターのうちで主要なものとしては、次のようなものがあります。

file `read` *int true* または *file* `read` *false* **オペレーター**

ファイルオブジェクト *file* に関連付けられたファイルから 1 個の文字を読み込んで、その文字コード（整数）をスタックにプッシュして、さらに真をプッシュします。文字を読み込む前に、すでに読み書き位置がファイルの終わりに到達している場合は、偽をプッシュします。

file string `readline` *substring bool* **オペレーター**

ファイルオブジェクト *file* に関連付けられたファイルから 1 個の行を読み込んで、その行（改行を示す文字は含みません）を文字列 *string* に上書きして、上書きされた *string* の部分文字列をスタックにプッシュして、さらに真偽値をプッシュします。真偽値は、行を読み込むことができた場合は真で、改行を示す文字を読み込む前に読み書き位置がファイルの終わりに到達した場合は偽です。

file string `readstring` *substring bool* **オペレーター**

ファイルオブジェクト *file* に関連付けられたファイルから文字列を読み込んで、その文字列を文字列 *string* に上書きして、上書きされた *string* の部分文字列をスタックにプッシュして、さらに真偽値をプッシュします。真偽値は、*string* と同じ長さの文字列を読み込むことができた場合は真で、読み書き位置がファイルの終わりに到達するまでに読み込んだ文字列の長さが *string* よりも短かった場合は偽です。

file `token` *any true* または *file* `token` *false* **オペレーター**

ファイルオブジェクト *file* に関連付けられたファイルから 1 個のトークンを読み込んで、そのトークンがあらわしているオブジェクトをスタックにプッシュして、さらに真をプッシュします。

ホワイトスペース以外の文字を読み込むことができないまま読み書き位置がファイルの終わりに到達した場合は、偽をスタックにプッシュします。

file int `write` −　　**オペレーター**

ファイルオブジェクト *file* に関連付けられたファイルに、整数 *int* を文字コードとする文字を書き込みます。

file string `writestring` −　　**オペレーター**

ファイルオブジェクト *file* に関連付けられたファイルに文字列 *string* を書き込みます。

2.13.6　ファイルを扱う手続き

それでは、ファイルを扱う手続きを定義してみましょう。

pathname `cat` −　　**手続き**

パス名 *pathname* で指定されたファイルの内容をモニターの画面に出力します。

プログラムの例　`cat.ps`

```
%!PS-Adobe-3.0
/cat {
    3 dict begin
    /pathname exch def
    /in pathname (r) file def
    /s 256 string def
    {
        in s readline not { pop exit } if
        =
    } loop
    in closefile
    end
} def
```

ファイルの例　`animal.txt`

```
armadillo
kangaroo
orangutan
```

実行例

```
GS>(cat.ps) run
GS>(animal.txt) cat
armadillo
kangaroo
orangutan
```

$pathname_1$ $pathname_2$ `copy` −　　**手続き**

パス名 $pathname_1$ で指定されたファイルの内容をパス名 $pathname_2$ で指定されたファイルにコピーします。$pathname_2$ で指定されるファイルが存在しない場合は、そのファイルを新しく生成します。

プログラムの例　`copy.ps`

```
%!PS-Adobe-3.0
/copy {
    4 dict begin
    /pathname2 exch def
    /pathname1 exch def
    /in pathname1 (r) file def
    /out pathname2 (w) file def
    {
        in read not { exit } if
        out exch write
    } loop
    in closefile
    out closefile
    end
} def
```

実行例

```
GS>(copy.ps) run
GS>(animal.txt) (animal2.txt) copy
```

2.13.7 標準入出力

オペレーティングシステムは、「標準入出力」(standard IO) と呼ばれる三つのファイルを持っています。標準入出力は、実体を持たない仮想的なファイルで、オペレーティングシステムによって特定のファイルに割り当てられることによって、実際に読み書きをすることが可能になります。

標準入出力を構成する三つのファイルは、それぞれ、「標準入力」(standard input)、「標準出力」(standard output) 「標準エラー」(standard error) と呼ばれます。通常、標準入力はキーボードに、標準出力と標準エラーはモニターに割り当てられていますが、それらの割り当てを別のファイルに切り替えることも可能です。標準入出力をキーボードやモニターから別のファイルに切り替えることを、標準入出力を「リダイレクトする」(redirect) と言います（名詞は「リダイレクション」(redirection) です)。

これまでに何度も登場している == や = や pstack は、標準出力に対する書き込みを実行するオペレーターです。標準出力にデータを書き込むオペレーターとしては、それら以外に、print というのもあります。

string `print` – **オペレーター**

文字列 *string* を標準出力に出力します。

== や = は出力の最後に改行を付け加えますが、print は、ただ単にポップした文字列をそのまま出力するだけです。したがって、print に改行を出力させるためには、文字列の中に改行を入れておく必要があります。

```
GS>(warabi) print
warabiGS>(warabi\n) print
warabi
GS>_
```

2.13.8 標準入出力のファイルオブジェクト

PostSctipt のインタプリタは、標準入出力に関連付けられたファイルオブジェクトを保持しています。それらのファイルオブジェクトは、標準入出力を指定するパス名を使って取得することができます。

標準入出力を指定するパス名というのは、次のような文字列です（ただし、これらのパス名が使えるのは PostScript のプログラムの中だけです)。

標準入力	`%stdin`
標準出力	`%stdout`
標準エラー	`%stderr`

これらのパス名とアクセス文字列をスタックにプッシュして file を実行すると、標準入出力に関連付けられたファイルオブジェクトがスタックにプッシュされます。たとえば、

```
/in (%stdin) (r) file def
```

というプログラムを実行することによって、in という名前を、標準入力からデータを読み込むためのファイルオブジェクトに束縛することができます。

```
GS>(copy.ps) run
GS>(animal.txt) (%stdout) copy
armadillo
kangaroo
orangutan
```

2.13.9 ファイルの削除とパス名の変更

次のオペレーターを使うことによって、ファイルを削除したり、ファイルのパス名を変更したりすることができます。

pathname `deletefile` – **オペレーター**

パス名 *pathname* で指定されたファイルを削除します。

pathname$_1$ *pathname*$_2$ `renamefile` − **オペレーター**

パス名 *pathname*$_1$ で指定されたファイルのパス名を *pathname*$_2$ に変更します。

第 3 章　ページ記述言語としての PostScript

3.1　グラフィックスの基礎知識

3.1.1　この節について

PostScript はページ記述言語の一種です。したがって、それを使うことによって、ページの上に印刷することのできるグラフィックスを記述することができます。

この節では、PostScript を使ってグラフィックスを記述する方法を理解するために必要となる基礎的な知識について説明したいと思います。

3.1.2　グラフィックス状態

グラフィックスに関連するさまざまな状態（たとえば、色や線幅など）をあらわしているデータの集合は、「グラフィックス状態」(graphics state) と呼ばれます。

グラフィックス状態は、そのコピーを、「グラフィックス状態スタック」(graphics state stack) と呼ばれるスタックに保存しておくことができます。ですから、ある時点で存在しているグラフィックス状態は、ひとつだけとは限りません。しかし、ある時点で操作することができるグラフィックス状態は、ひとつだけです。グラフィックス状態のうちで、現在の時点で操作の対象になっているものは、「カレントグラフィックス状態」(current graphics state) と呼ばれます。

3.1.3　カレントページ

PostScript のプログラムは、「カレントページ」(current page) と呼ばれる平面の上にグラフィックスを描画します。

カレントページは、物理的に存在する平面ではなくて、PostScript のインタプリタによって作り出された仮想的な平面です。PostScript のプログラムは、カレントページにグラフィックスを描画したのち、`showpage` というオペレーターを実行することによって、そのグラフィックスを、プリンターやモニターなどの物理的な装置に転送することができます。

− `showpage` − **オペレーター**

カレントページに描画されたグラフィックスを物理的な装置に転送します。そして、カレントページに描画されたグラフィックスを消去します。

3.1.4　座標系

カレントページの上に描画されるグラフィックスを記述するためには、点の位置を記述するということが必要になります。

空間の中にある点の位置は、通常、「座標系」(coordinate system) と呼ばれるものを使って記述されます。カレントページの上の点も、座標系を使うことによって記述することができます。

座標系は、「軸」(axis) と呼ばれる、方向を持つ直線から構成されます。ひとつの座標系を構成しているすべての軸は、「原点」(origin) と呼ばれる点で交わっています。

軸の本数は、空間の次元と同じだけ必要です。したがって、平面（2 次元空間）の座標系は、2 本の軸から構成されることになります。それらの軸は、通常、「x 軸」(x axis) と「y 軸」(y axis) と呼ばれます。

空間の中の点の位置は、「座標」(coordinate) と呼ばれる数値の列によって記述されます。座標は、原点を基準とする、それぞれの軸の方向への距離を示す数値から構成されています。

平面（2 次元空間）の中の点の位置は、x 軸方向の距離 x と y 軸方向の距離 y から構成される、(x, y) という数値の列によって記述されます。x 軸方向の距離は「x 座標」(x coordinate) と呼ばれ、y 軸方向の距離は「y 座標」(y coordinate) と呼ばれます。

3.1.5　デフォルトの座標系

グラフィックス状態は、カレントページの座標系に関する次のデータを持っています。

- 原点の位置。

- 軸の方向。
- 長さの単位（x 軸方向、y 軸方向）。

これらのデータは、オペレーターを実行することによって、変更することが可能です。これらのデータに関して変更が何も加わっていない座標系、つまり初期状態の座標系は、「デフォルトの座標系」(default coordinate system) と呼ばれます。

デフォルトの座標系は、次のような状態になっています。

- 原点の位置は、カレントページの左下の隅。
- x 軸の方向は右、y 軸の方向は上。
- 長さの単位は、x 軸方向も y 軸方向も、72 分の 1 インチ。

3.1.6 ポイント

先ほど述べたとおり、デフォルトの座標系では、長さの単位が 72 分の 1 インチになっています。つまり、プログラムの中に 1 という長さを書いたとするならば、それは 72 分の 1 インチを意味することになるわけです。

PostScript では、この 72 分の 1 インチという長さは、「ポイント」(point) と呼ばれます。つまり、ポイントというのは、

1 ポイント = 72 分の 1 インチ = 約 0.352778 ミリメートル

という単位のことです。ただし、「ポイント」と呼ばれる単位には、PostScript で使われているもののほかに、

ディドー式ポイント (Didot point)	1 ポイント = 約 0.3759 ミリメートル
アメリカ式ポイント (Anglo-American point)	1 ポイント = 約 0.3514 ミリメートル

というのもあります。PostScript で使われている「ポイント」は、それらと区別するために、「PostScript ポイント」(PostScript point)、「DTP ポイント」(DTP point)、「コンピュータポイント」(computer point) などと呼ばれることもあります。

1 ミリメートルは約 2.834646 ポイントですので、ミリメートルをポイントに換算する、

```
/mm { 2.834646 mul } def
```

という手続きを定義しておくことによって、プログラムの中で長さをミリメートルで記述することができるようになります。たとえば、`7 mm` と書くことによって、7 ミリメートルをポイントに換算した数値をスタックにプッシュすることができます。

3.2 パス

3.2.1 パスとは何か

グラフィックス状態は、「パス」(path) と呼ばれるデータを含んでいます。パスというのは、カレントページの上に作られた何本かの線の集合（またはそれをあらわしているデータ）のことです。

カレントグラフィックス状態が持っているパスは、「カレントパス」(current path) と呼ばれます。

パスを初期化したり、パスに対して線を追加したりすることを、パスを「構築する」(construct) と言います。パスを構築するために使われるオペレーターは、「パス構築オペレーター」(path construction operator) と呼ばれます。

パスに対して線を追加するためには、まず最初に、`newpath` というパス構築オペレーターを実行することによって、そのパスを初期化する必要があります。

− `newpath` − **オペレーター**

カレントパスを初期化します。

3.2.2 カレントポイント

パスに対する線の追加は、カレントページの上で仮想的なペンを動かすことによって実行されます。

仮想的なペンは、ページに接触した状態で移動させることもできますし、ページから離れた状態で移動させることもできます。ページに接触した状態でペンを移動させた場合には、その移動の軌跡がカレントパスに追加されます。逆に、ページから離れた状態でペンを移動させた場合には、カレントパスには何も追加されません。

グラフィックス状態は、仮想的なペンの位置というデータを持っています。仮想的なペンの位置は、「ペン位置」(pen location) と呼ばれます。そして、カレントグラフィックス状態が持っているペン位置は、「カレントポイント」(current point) と呼ばれます。

moveto または rmoveto というパス構築オペレーターを使うことによって、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ということ（つまり、ページから離れた状態でペンを移動させるということ）ができます。moveto と rmoveto との相違点は、移動先の位置を指定する方法が絶対的（absolute）なのか相対的（relative）なのかというところにあります。moveto は絶対的で、rmoveto は相対的です。

x y moveto – **オペレーター**

カレントポイントを (x, y) という位置へ移動させます。たとえば、

```
400 300 moveto
```

というプログラムを実行すると、カレントポイントは $(400, 300)$ という位置へ移動します。

dx dy rmoveto – **オペレーター**

カレントポイントを、x 軸の方向へ dx、y 軸の方向へ dy だけ移動させます。つまり、カレントポイントが (x, y) だとするならば、移動先の座標は $(x + dx, y + dy)$ になるということです。たとえば、カレントポイントが $(400, 300)$ だとするときに、

```
70 -20 rmoveto
```

というプログラムを実行すると、カレントポイントは $(470, 280)$ という位置へ移動します。

3.2.3　カレントポイントの生成

newpath を使って初期化された直後のパスは、カレントポイントが存在していない状態になっています。

カレントポイントは、moveto を使うことによって生成することができます。moveto は、カレントポイントが存在していない場合、指定された位置にカレントポイントを生成します。

ちなみに、rmoveto には、カレントポイントを生成するという機能はありません。

3.2.4　サブパス

ひとつのパスを構成している線のうちで、ひとつに連結されているそれぞれの部分（つまり一筆で書かれているそれぞれの部分）は、そのパスの「サブパス」(subpath) と呼ばれます。パスは、1 個以上のサブパスから構成されます。

moveto または rmoveto を実行するということは、新しいサブパスを開始するという意味を持っていると考えることができます。

ひとつのパスを構成しているサブパスのうちで、現時点で線が追加される対象となっているものは、そのパスの「カレントサブパス」(current subpath) と呼ばれます。

3.2.5　描画

パスを構成しているのは、目には見えない線です。パスに基づいて、目に見えるグラフィックスをカレントページの上に描くことを、パスを「描画する」(paint) と言います。

パスを描画するという動作には、「線を描画する」(stroke) という動作と、「領域を塗りつぶす」(fill) という動作の二種類があります。

3.2.6　グラフィックスを出力する手順

これまで説明してきたことを整理して、プリンターやモニターにグラフィックスを出力するために実行しないといけないことを、時間の順序に沿って並べてみると、次のようになります。

(1) newpath で、カレントパスを初期化する。
(2) カレントパスに線を何本か追加する。
(3) カレントパスをカレントページに描画する。

(4) showpage で、カレントページを物理的な装置に転送する。

ただし、(1) から (3) までの操作は、(4) を実行する前に何回でも繰り返すことが可能です。

3.3 直線

3.3.1 カレントパスへの直線の追加

この節では、直線 (line) の描画に関連する問題について説明したいと思います。

lineto または rlineto というパス構築オペレーターを使うことによって、カレントパスに直線を追加することができます。これらのオペレーターは、ペンとカレントページとを接触させた状態で、カレントポイントから指定された位置まで、直線に沿ってペンを移動させます。移動先を指定する方法は、lineto の場合は絶対的で、rlineto の場合は相対的です。

x y lineto – **オペレーター**

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) へ移動させます。

たとえば、カレントポイントが $(400, 300)$ だとするとき、

```
500 600 lineto
```

というプログラムを実行すると、$(400, 300)$ と $(500, 600)$ とをつなぐ直線がカレントパスに追加されて、カレントポイントは $(500, 600)$ へ移動します。

dx dy rlineto – **オペレーター**

カレントポイントが (x, y) だとするとき、カレントポイントと $(x + dx, y + dy)$ 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 $(x + dx, y + dy)$ へ移動させます。

たとえば、カレントポイントが $(400, 300)$ だとするとき、

```
70 -20 rlineto
```

というプログラムを実行すると、$(400, 300)$ と $(470, 280)$ とをつなぐ直線がカレントパスに追加されて、カレントポイントは $(470, 280)$ へ移動します。

3.3.2 カレントパスに沿った線の描画

stroke というオペレーターを使うことによって、カレントパスに沿った線をカレントページの上に描画することができます。

– stroke – **オペレーター**

カレントパスに沿った線をカレントページの上に描画します。

プログラムの例 lineto.ps

```
%!PS-Adobe-3.0
newpath
100 500 moveto
300 700 lineto
500 500 lineto
300 300 lineto
stroke
showpage
```

次のプログラムは、上のプログラムと同じ図形を描画するのですが、直線をカレントパスに追加するためのオペレーターとして、lineto ではなくて rlineto を使っています。

プログラムの例 rlineto.ps

```
%!PS-Adobe-3.0
newpath
100 500 moveto
200 200 rlineto
200 -200 rlineto
-200 -200 rlineto
stroke
showpage
```

3.3.3 線幅

グラフィックス状態が持っている状態にひとつに、「線幅」(line width) と呼ばれるものがあります。

線幅というのは線の太さ (thickness) のことで、数値によってあらわされます。デフォルトの線幅は 1 です（座標系の拡大率がデフォルトのままだとすれば、1 ポイントということです)。

線幅は、`setlinewidth` というオペレーターを実行することによって設定することができます。

num `setlinewidth` – **オペレーター**

カレントグラフィックス状態に対して、線幅として数値 *num* を設定します。

プログラムの例 `width.ps`

```
%!PS-Adobe-3.0
/hline {
    2 dict begin
    /width exch def
    /y exch def
    newpath
    100 y moveto
    400 0 rlineto
    width setlinewidth
    stroke
    end
}def
/seventeenlines {
    3 dict begin
    /width 4 def
    100 40 740 {
        /y exch def
        y width hline
        /width width 2 add def
    } for
    end
} def
seventeenlines
showpage
```

このプログラムは、線幅の異なる 17 本の水平な直線を描画します。線幅は、一番下が 4 ポイントで、上に向かって 2 ポイントずつ太くなっていきます。

3.3.4 開いたサブパスと閉じたサブパス

サブパスは、「開いたサブパス」(open subpath) と「閉じたサブパス」(closed subpath) の 2 種類に分類することができます。

開いたサブパスというのは折れ線 (polyline) のことで、閉じたサブパスというのは多角形 (polygon) のことです。つまり、開いたサブパスと閉じたサブパスとの相違点は、前者には始まりの点と終わりの点があるのに対して、後者には始まりの点も終わりの点もないということです。

ちなみに、開いたサブパスが持っている始まりの点と終わり点のそれぞれは、「端点」(cap) と呼ばれます。そして、端点ではない点、つまり線と線とがそこで接続されている点は、「接続点」(join) と呼ばれます。

閉じたサブパスを作るためには、サブパスを「閉じる」(close) という操作をする必要があります。サブパスを閉じたいときは、`closepath` というオペレーターを使います。

– `closepath` – **オペレーター**

カレントポイントからカレントサブパスの開始点までの直線をカレントサブパスに追加して、カレントサブパスを閉じます。

`closepath` を実行すると、カレントサブパスの構築は終了しますので、それ以降に追加される線は新しいサブパスを構成することになります。

プログラムの例 `close.ps`

```
%!PS-Adobe-3.0
newpath
200 400 moveto
200 0 rlineto
```

```
0 -200 rlineto
-200 0 rlineto
0 200 rlineto
200 700 moveto
200 0 rlineto
0 -200 rlineto
-200 0 rlineto
closepath
60 setlinewidth
stroke
showpage
```

このプログラムは、二つの正方形を描画します。下の正方形は開いたサブパスによって描画されたもので、上の正方形は閉じたサブパスによって描画されたものです。

3.3.5 塗りつぶし

第 3.2 節で説明したように、パスを描画するという動作には、「線を描画する」(stroke) という動作と、「領域を塗りつぶす」(fill) という動作の二種類があります。

stroke は、カレントパスに沿って線を描画するという動作を実行するオペレーターです。それに対して、fill というオペレーターは、カレントパスによって囲まれている領域を塗りつぶすという動作をします。

— fill — **オペレーター**

カレントパスを構成しているそれぞれのサブパスについて、それによって囲まれている、カレントページの上の領域を塗りつぶします。開いたサブパスについては、暗黙のうちにそれを閉じたのちに、その内部を塗りつぶします。

プログラムの例 fill.ps

```
%!PS-Adobe-3.0
newpath
100 400 moveto
200 -200 rlineto
200 200 rlineto
100 500 moveto
200 200 rlineto
200 -200 rlineto
fill
showpage
```

このプログラムは、2 個のサブパスから構成されるパスを構築したのち、fill を実行することによって、それぞれのサブパスによって囲まれている領域を塗りつぶしています。

3.3.6 領域の内部性の判定

領域の内部であるという性質は、その領域の「内部性」(insideness) と呼ばれます。

PostScript は、領域の内部性を判定するための規則として次の二種類のものをサポートしています。

- ワインディング規則 (nonzero winding number rule)
- 偶奇規則 (even-odd rule)

領域を塗りつぶすオペレーターとしては、fill のほかに eofill というのもあります。これらは、ほとんど同じ動作をするオペレーターで、相違点は領域の内部性を判定する規則だけです。fill がワインディング規則によって領域の内部性を判定するのに対して、eofill は偶奇規則によってそれを判定します。

— eofill — **オペレーター**

カレントパスを構成しているそれぞれのサブパスについて、それによって囲まれている、カレントページの上の領域を塗りつぶします。開いたサブパスについては、暗黙のうちにそれを閉じたのちに、その内部を塗りつぶします。領域の内部性は偶奇規則によって判定します。

プログラムの例 eofill.ps

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

```
/star {
    2 dict begin
    /y exch def
    /x exch def
    newpath
    x y moveto
    300 0 rlineto
    -250 -200 rlineto
    100 300 rlineto
    100 -300 rlineto
    end
} def
0 0.4 0 setrgbcolor
150 650 star
fill
150 300 star
eofill
showpage
```

このプログラムは、星形のパスを構築してその内部の領域を塗りつぶすという動作を、二回、実行します。上の位置に描画される星形は `fill` で塗りつぶしたもので、下の位置に描画される星形は `eofill` で塗りつぶしたものです。

3.4 色

3.4.1 色の基礎

グラフィックス状態は、1 個の色のデータを持っています。カレントグラフィックス状態が持っている色のデータは、「カレントカラー」(current color) と呼ばれます。

`stroke` や `fill` などを使ってカレントページにグラフィックスを描画するときは、そのグラフィックスの色としてカレントカラーが使われます。

色は、数値の組み合わせを使って記述することができます。数値の組み合わせによって色を記述する方法は、「色空間」(color space) と呼ばれます。色空間にはさまざまなものがありますが、PostScript で使うことができるのは次の三つです。

- 加法混色 (additive mixture of colors)
- 減法混色 (subtractive mixture of colors)
- グレイレベル (gray level)

3.4.2 加法混色

赤 (red)、緑 (green)、青 (blue) という光の三原色の比率によって色を記述する色空間は、「加法混色」(additive mixture of colors)、または、三原色の頭文字から、「RGB」と呼ばれます。

`setrgbcolor` というオペレーターを使うことによって、加法混色によって記述された色をカレントカラーに設定することができます。

red green blue `setrgbcolor` – **オペレーター**

red を赤の比率、*green* を緑の比率、*blue* を青の比率とする加法混色によって記述された色をカレントカラーに設定します。それぞれの比率は、0 から 1 までの数値で指定します。

プログラムの例　`rgb.ps`

```
%!PS-Adobe-3.0
/hline {
    5 dict begin
    /blue exch def
    /green exch def
    /red exch def
    /y exch def
    /x exch def
    newpath
    x y moveto
    30 0 rlineto
    30 setlinewidth
```

```
    red green blue setrgbcolor
    stroke
    end
}def
/hundredlines {
    4 dict begin
    /red 0 def
    300 40 660 {
        /y exch def
        /green 0 def
        100 40 460 {
            /x exch def
            x y red green 0 hline
            /green green 0.1 add def
        } for
        /red red 0.1 add def
    } for
    end
} def
hundredlines
showpage
```

このプログラムは、加法混色の赤と緑の比率を変化させながら 100 本の直線を描画します。青の比率は常に 0 です。

3.4.3 減法混色

シアン (cyan)、マゼンタ (magenta)、イエロー (yellow) という色料の三原色の比率と黒 (black) の比率によって色を記述する色空間は、「減法混色」(subtractive mixture of colors)、または、三原色の頭文字と「色調」(key tone) から、「CMYK」[1] と呼ばれます。

`setcmykcolor` というオペレーターを使うことによって、減法混色によって記述された色をカレントカラーに設定することができます。

cyan magenta yellow black `setcmykcolor` – **オペレーター**

cyan をシアンの比率、*magenta* をマゼンタの比率、*yellow* をイエローの比率、*black* を黒の比率とする減法混色によって記述された色をカレントカラーに設定します。それぞれの比率は、0 から 1 までの数値で指定します。

プログラムの例 `cmyk.ps`

```
%!PS-Adobe-3.0
/hline {
    6 dict begin
    /black exch def
    /yellow exch def
    /magenta exch def
    /cyan exch def
    /y exch def
    /x exch def
    newpath
    x y moveto
    30 0 rlineto
    30 setlinewidth
    cyan magenta yellow black setcmykcolor
    stroke
    end
}def
/hundredlines {
    4 dict begin
    /cyan 0 def
    300 40 660 {
        /y exch def
        /magenta 0 def
        100 40 460 {
            /x exch def
```

[1] K は、black の最後の文字と説明されることもあります。

```
            x y cyan magenta 0 0 hline
            /magenta magenta 0.1 add def
        } for
        /cyan cyan 0.1 add def
    } for
    end
} def
hundredlines
showpage
```

このプログラムは、減法混色のシアンとマゼンタの比率を変化させながら 100 本の直線を描画します。イエローと黒の比率は常に 0 です。

3.4.4 グレイレベル

グレイ (gray) の明るさによって色を記述する色空間は、「グレイレベル」(gray level) と呼ばれます。

setgray というオペレーターを使うことによって、グレイレベルによって記述された色をカレントカラーに設定することができます。

num setgray – **オペレーター**

num を明るさとするグレイをカレントカラーに設定します。明るさは、0 から 1 までの数値で指定します。0 は黒 (black) になって、1 は白 (white) になります。

プログラムの例 gray.ps

```
%!PS-Adobe-3.0
/dline {
    newpath
    150 100 moveto
    300 600 rlineto
    40 setlinewidth
    stroke
} def
/hline {
    2 dict begin
    /gray exch def
    /y exch def
    newpath
    100 y moveto
    400 0 rlineto
    30 setlinewidth
    gray setgray
    stroke
    end
}def
/elevenlines {
    3 dict begin
    /gray 0 def
    200 40 600 {
        /y exch def
        y gray hline
        /gray gray 0.1 add def
    } for
    end
} def
dline
elevenlines
showpage
```

このプログラムは、1 本の斜めの直線を描画したのち、グレイの明るさを変化させながら 11 本の直線を描画します。

このプログラムを実行した結果から分かるとおり、カレントページに対してグラフィックスを描画すると、そのグラフィックスは、それ以前に描画されたグラフィックスの上に重なります。

3.5 線の形状

3.5.1 線の形状の基礎

グラフィックス状態は、stroke によって描画される線の形状についてのデータを持っています。

グラフィックス状態が持っている線の形状についてのデータとしては、次のようなものがあります。

- 端点の形状
- 接続点の形状
- マイターリミット
- 破線配列
- 破線オフセット

3.5.2 端点の形状

第 3.3 節でも説明しましたが、開いたサブパスが持っている始まりの点と終わり点のそれぞれは、「端点」(cap) と呼ばれます。グラフィックス状態は、端点の形状を示す整数を持っています。

setlinecap というオペレーターを使うことによって、カレントグラフィックス状態に対して、端点の形状を示す整数を設定することができます。

int setlinecap – **オペレーター**

カレントグラフィックス状態に対して、端点の形状を示す整数として *int* を設定します。整数は、0、1、2 のいずれかで、それぞれの整数は次のような形状を意味しています。

0 バットキャップ (butt cap)。端点を通る、線に対して垂直な直線で、線を切断した形状。

1 ラウンドキャップ (round cap)。端点を中心とする、線幅を直径とする半円を付加した形状。

2 スクエアキャップ (projecting square cap)。バットキャップと同じように、線に対して垂直な直線で切断した形状ですが、線幅の半分だけ線を延長したところで切断します。

プログラムの例 cap.ps

```
%!PS-Adobe-3.0
/hline {
    2 dict begin
    /cap exch def
    /y exch def
    newpath
    150 y moveto
    300 0 rlineto
    100 setlinewidth
    0.5 1 1 setrgbcolor
    cap setlinecap
    stroke
    newpath
    150 y moveto
    300 0 rlineto
    1 setlinewidth
    1 0 0 setrgbcolor
    0 setlinecap
    stroke
    end
}def
650 0 hline
500 1 hline
350 2 hline
showpage
```

3.5.3 接続点の形状

第 3.3 節でも説明しましたが、端点ではない点、つまり線と線とがそこで接続されている点は、「接続点」(join) と呼ばれます。グラフィックス状態は、接続点の形状を示す整数を持っています。

setlinejoin というオペレーターを使うことによって、カレントグラフィックス状態に対して、接続点の形状を示す整数を設定することができます。

int setlinejoin – **オペレーター**

カレントグラフィックス状態に対して、接続点の形状を示す整数として *int* を設定します。整数は、0、1、2 のいずれかで、それぞれの整数は次のような形状を意味しています。

0 マイタージョイン (miter join)。線の外側の輪郭を、それらが交わるまで直線で延長した形状。

1 ラウンドジョイン (round join)。線の外側の輪郭が、接続点を中心とする、線幅を直径とする円を描く形状。

2 ベベルジョイン (bevel join)。両側の線を端点としてバットキャップで描画したときにできる三角形のくぼみを埋めた形状。

プログラムの例 join.ps

```
%!PS-Adobe-3.0
/harpoon {
    2 dict begin
    /join exch def
    /y exch def
    newpath
    150 y moveto
    250 0 rlineto
    -150 120 rlineto
    80 setlinewidth
    0.5 1 1 setrgbcolor
    join setlinejoin
    stroke
    newpath
    150 y moveto
    250 0 rlineto
    -150 120 rlineto
    1 setlinewidth
    1 0 0 setrgbcolor
    0 setlinejoin
    stroke
    end
}def
600 0 harpoon
400 1 harpoon
200 2 harpoon
showpage
```

3.5.4 マイターリミット

マイタージョインという接続点の形状は、線が十分に大きな角度で折れ曲がっている場合は問題がないのですが、きわめて小さな角度で折れ曲がっている場合、先端が長く延びて不自然になります。ですから、PostScript のインタプリタは、接続点の形状としてマイタージョインが設定されていたとしても、角度が限度を超えて小さい場合には、ベベルジョインで接続点を描画します。

通常の線幅に対する接続点の線幅の比率は、「マイター長」(miter length) と呼ばれます。そして、それを超えるとベベルジョインになるマイター長は、「マイターリミット」(miter limit) と呼ばれます。マイターリミットは、グラフィックス状態が持っているデータのひとつです。

マイタージョインは、マイターリミットが小さければ小さいほど、大きな角度でベベルジョインに切り替わります。マイターリミットのデフォルトは 10 で、この場合、ベベルジョインに切り替わるのは、角度が約 11 度よりも小さくなったときです。マイターリミットが 2 の場合は、60 度よりも小さくなったときにベベルジョインに切り替わります。

setmiterlimit というオペレーターを使うことによって、カレントグラフィックス状態に対してマイターリミットを設定することができます。

num setmiterlimit – **オペレーター**

カレントグラフィックス状態に対して、マイターリミットとして数値 *num* を設定します。*num* は 1 以上でないといけません。

プログラムの例 miter.ps

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

```
/harpoon {
    2 dict begin
    /miterlimit exch def
    /y exch def
    newpath
    150 y moveto
    250 0 rlineto
    -150 120 rlineto
    80 setlinewidth
    0 0.4 0 setrgbcolor
    0 setlinejoin
    miterlimit setmiterlimit
    stroke
    end
}def
600 10 harpoon
400 2 harpoon
showpage
```

このプログラムは、二つのパスを描画します。接続点の形状としては、どちらもマイタージョインを設定しているのですが、下のパスは、マイターリミットとして 2 を設定して描画していますので、接続点の形状がベベルジョインになっています。

3.5.5 破線配列と破線オフセット

線と切れ目を交互に繰り返す線は「破線」(dashed line) と呼ばれ、まったく切れ目を持たない線は「実線」(solid line) と呼ばれます。そして、線と切れ目から構成されるパターンは、破線パターン (dash pattern) と呼ばれます。切れ目を含まない破線パターンで線を描画すると、実線になります。

破線パターンは、「破線配列」(dash array) と呼ばれる配列によってあらわされます。破線配列は、線の長さと切れ目の長さを交互に並べたものです。

長さが 0 の配列、つまり空配列は、切れ目を含まない破線パターン、つまり実線の破線パターンをあらわす破線配列です。

偶数個の数値から構成される配列は、奇数個目の数値を線の長さ、偶数個目の数値を切れ目の長さとする破線パターンをあらわす破線配列です。たとえば、

```
[ 30 10 ]
```

という配列は、長さが 30 の線と長さが 10 の切れ目を交互に繰り返す破線パターンをあらわします。

奇数個の数値から構成されている配列は、その数値の列を二回並べた配列と同一の破線パターンをあらわします。たとえば、

```
[ 20 ]
```

という配列は、

```
[ 20 20 ]
```

という配列と同一の破線パターンをあらわしています。同じように、

```
[ 10 20 30 ]
```

という配列は、

```
[ 10 20 30 10 20 30 ]
```

という配列と同一の破線パターンをあらわしています。

破線は、破線パターンの途中から開始することも可能です。破線パターンの先頭から破線を開始する位置までの長さは、「破線オフセット」(dash offset) と呼ばれます。

グラフィックス状態は破線配列と破線オフセットを持っていて、`stroke` は、それらを使って線を描画します。デフォルトでは、破線配列として空配列が設定されていますので、実線が描画されることになります。

`setdash` というオペレーターを使うことによって、カレントグラフィックス状態に対して破線配列と破線オフセットを設定することができます。

array offset setdash　−　　**オペレーター**

カレントグラフィックス状態に対して、破線配列として *array* を設定して、破線オフセットとして *offset* を設定します。*array* は、マイナスではない数値から構成されていないといけません。また、すべての要素が 0 であってはいけません。

プログラムの例　dash.ps

```
%!PS-Adobe-3.0
/vline {
    newpath
    100 150 moveto
    100 750 lineto
    500 150 moveto
    500 750 lineto
    1 0 0 setrgbcolor
    stroke
} def
/hline {
    3 dict begin
    /offset exch def
    /array exch def
    /y exch def
    newpath
    100 y moveto
    400 0 rlineto
    0 0.4 0.8 setrgbcolor
    40 setlinewidth
    array offset setdash
    stroke
    end
}def
vline
700 [] 0 hline
600 [ 30 10 ] 0 hline
500 [ 20 ] 0 hline
400 [ 10 20 30 ] 0 hline
300 [ 50 10 10 10 ] 0 hline
200 [ 50 10 10 10 ] 20 hline
showpage
```

3.5.6　破線の端点と接続点の形状

グラフィックス状態が持っている端点と接続点の形状は、破線を描画する場合にも適用されます。また、端点の形状は、破線を構成している個々の線に対しても適用されます。

プログラムの例　roudash.ps

```
%!PS-Adobe-3.0
newpath
100 200 moveto
400 0 rlineto
-200 500 rlineto
closepath
60 setlinewidth
0.4 0.8 0 setrgbcolor
1 setlinecap
1 setlinejoin
[200 80] 0 setdash
stroke
showpage
```

3.6　円弧

3.6.1　円弧の基礎

この節では、円弧 (arc) の描画に関連する問題について説明したいと思います。

カレントパスに円弧を追加する方法としては、次の 2 種類のものがあります。

- 中心を指定する方法。
- 接線を指定する方法。

3.6.2 中心の指定による円弧

arc またはarcn というパス構築オペレーターは、オペランドで指定された点を中心とする円弧をカレントパスに追加します。

$x\ y\ r\ ang_1\ ang_2$ arc − **オペレーター**

(x, y) が中心で、半径が r で、開始角度が ang_1 で、終了角度が ang_2 で、回転の方向が反時計回りの円弧を、カレントパスに追加します。

角度は、x 軸の方向が 0 度で、反時計回りで大きくなります。

$x\ y\ r\ ang_1\ ang_2$ arcn − **オペレーター**

arc とほとんど同じ動作をします。違いは回転の方向だけです。 arc が反時計回りなのに対して、 arcn は時計回りです。

プログラムの例 arc.ps

```
%!PS-Adobe-3.0
newpath
300 600 100 0 225 arc
stroke
300 400 100 0 225 arcn
stroke
showpage
```

3.6.3 円弧とカレントポイント

arc またはarcn でカレントパスに円弧を追加する場合、カレントポイントは、存在していても存在していなくてもどちらでもかまいません。

カレントポイントが存在するときにarc またはarcn が実行された場合、カレントポイントと円弧の開始点とをつなぐ直線がカレントパスに追加されます。そして、カレントパスは、円弧の終了点へ移動します。

カレントポイントが存在しないときにarc またはarcn が実行された場合、円弧の終了点にカレントポイントが生成されます。

プログラムの例 kofun.ps

```
%!PS-Adobe-3.0
newpath
200 300 moveto
300 540 100 230 310 arcn
400 300 lineto
closepath
stroke
showpage
```

3.6.4 接線の指定による円弧

直線と直線とを円弧で接続したい、ということがしばしばあります。この場合の円弧の作り方としては、中心を指定する作り方よりも、それらの直線を接線として指定する作り方のほうが簡単です。

arct というパス構築オペレーターは、オペランドで指定された直線を接線とする円弧をカレントパスに追加します。

$x_1\ y_1\ x_2\ y_2\ r$ arct − **オペレーター**

カレントポイントと (x_1, y_1) とをつなぐ直線と、(x_1, y_1) と (x_2, y_2) とをつなぐ直線を接線とする、半径が r の円弧をカレントパスに追加して、カレントポイントを円弧の終了点へ移動させます。カレントポイントと円弧の開始点とが同一ではない場合は、それらをつなぐ直線をカレントパスに追加したのち、円弧を追加します。

プログラムの例 arct.ps

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

```
200 200 moveto
500 600 100 600 120 arct
0 1 0.8 setrgbcolor
30 setlinewidth
stroke
200 200 moveto
500 600 lineto
100 600 lineto
1 setlinewidth
1 0 0 setrgbcolor
stroke
showpage
```

3.7 ベジェ曲線

3.7.1 ベジェ曲線の基礎

カレントパスに追加することのできる線は、直線と円弧だけではありません。「ベジェ曲線」(Bézier curve) と呼ばれる、なめらかに曲がった線をカレントパスに追加する、ということも可能です。

ベジェ曲線の形は、開始点と終了点、そして、「制御点」(control point) と呼ばれる 2 個の点、という 4 個の点によって指定されます。2 個の制御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、「制御点 1」、「制御点 2」と呼ぶことにしましょう。ベジェ曲線は、まず開始点からスタートして、開始点と制御点 1 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、制御点 2 と終了点とをつなぐ直線に接する形で終了点に到達します。

3.7.2 カレントパスへのベジェ曲線の追加

curveto というパス構築オペレーターを使うことによって、ベジェ曲線をカレントパスに追加することができます。

$x_1\ y_1\ x_2\ y_2\ x_3\ y_3$ curveto − **オペレーター**

カレントポイントを開始点、(x_1, y_1) を制御点 1、(x_2, y_2) を制御点 2、(x_3, y_3) を終了点とするベジェ曲線をカレントパスに追加して、カレントポイントを (x_3, y_3) へ移動させます。

プログラムの例 bezier.ps

```
%!PS-Adobe-3.0
/bezier {
    8 dict begin
    /y3 exch def
    /x3 exch def
    /y2 exch def
    /x2 exch def
    /y1 exch def
    /x1 exch def
    /y0 exch def
    /x0 exch def
    newpath
    x0 y0 moveto
    x1 y1 x2 y2 x3 y3 curveto
    0 0.8 1 setrgbcolor
    40 setlinewidth
    stroke
    newpath
    x0 y0 moveto
    x1 y1 lineto
    x2 y2 moveto
    x3 y3 lineto
    1 0 0 setrgbcolor
    1 setlinewidth
    stroke
    end
} def
140 450 100 650 500 750 300 450 bezier
```

```
140 150 100 350 300 150 500 450 bezier
showpage
```

3.7.3 ベジェ曲線の連結

いくつかのベジェ曲線を連結することによって、全体としてなめらかに続いている複雑な曲線を作ることができます。ただし、ベジェ曲線をなめらかに連結するためには、それらの連結点と、その前後にある制御点、という 3 個の点が、1 本の直線の上に乗っていないといけません。

プログラムの例 bezbez.ps

```
%!PS-Adobe-3.0
/bezbez {
    14 dict begin
    /y6 exch def
    /x6 exch def
    /y5 exch def
    /x5 exch def
    /y4 exch def
    /x4 exch def
    /y3 exch def
    /x3 exch def
    /y2 exch def
    /x2 exch def
    /y1 exch def
    /x1 exch def
    /y0 exch def
    /x0 exch def
    newpath
    x0 y0 moveto
    x1 y1 x2 y2 x3 y3 curveto
    x4 y4 x5 y5 x6 y6 curveto
    0 1 0.8 setrgbcolor
    30 setlinewidth
    stroke
    newpath
    x0 y0 moveto
    x1 y1 lineto
    x2 y2 moveto
    x3 y3 lineto
    1 0 0 setrgbcolor
    1 setlinewidth
    stroke
    newpath
    x3 y3 moveto
    x4 y4 lineto
    x5 y5 moveto
    x6 y6 lineto
    0 0 1 setrgbcolor
    stroke
    end
} def
150 600 100 750 350 700 250 600 300 450 500 400 400 600 bezbez
150 300 100 450 350 400 250 300 100 150 500 100 400 300 bezbez
showpage
```

このプログラムは、二つのベジェ曲線を連結した線を、上下に二つ、描画します。下の線は、ベジェ曲線の連結点とその前後の制御点が 1 本の直線の上に乗っていますので、その全体がなめらかな曲線になっています。しかし、上の線は、それらの点が 1 本の直線の上に乗っていませんので、途中で折れ曲がっています。

3.8 文字列の描画

3.8.1 フォント

PostScript のプログラムは、カレントページに文字列を図形として描画することができます。

一般に、文字列を図形として描画するためには、それに含まれているすべての文字の形状が必要になります。文字の形状の集合は、「フォント」(font) と呼ばれます。

PostScript では、辞書の形式でフォントを記述したデータを使って、文字列を図形として描画します。フォントを記述した辞書は、「フォント辞書」(font dictionary) と呼ばれます。

PostScript のプログラムが文字列を描画するためには、それに先立って、そのために使われるフォント辞書の準備をする必要があります。フォント辞書は、次の 3 段階の手順を実行することによって準備することができます。

(1) フォント名によって識別されるフォント辞書を取得する。
(2) 大きさを拡大したフォント辞書を作成する。
(3) グラフィックス状態に対してフォント辞書を設定する。

3.8.2　フォント辞書の取得

フォント辞書は、「フォント名」(font name) と呼ばれる名前によって識別されます。たとえば、Times Roman と呼ばれる欧米語のフォントを記述したフォント辞書は、

```
Times-Roman
```

というフォント名によって識別されます。

`findfont` というオペレーターを使うことによって、フォント名によって識別されるフォント辞書を取得することができます。

name `findfont` *font*　　**オペレーター**

フォント名 *name* によって識別されるフォント辞書を取得して、それをオペランドスタックにプッシュします。

たとえば、

```
/Times-Roman findfont
```

というプログラムを実行することによって、`Times-Roman` というフォント名によって識別されるフォント辞書をオペランドスタックにプッシュすることができます。

3.8.3　フォント辞書の拡大

フォント辞書を構成している文字の垂直方向の長さを、そのフォント辞書の「大きさ」(size) と言います。

`findfont` によって取得されるフォント辞書は、大きさが 1 ポイントになっています。それ以外の大きさで文字を描画するためには、取得したフォント辞書を拡大したものを作成する必要があります。

`scalefont` というオペレーターを使うことによって、大きさを拡大したフォント辞書を作成することができます。

$font_1$ *scale* `scalefont` $font_2$　　**オペレーター**

フォント辞書 $font_1$ を *scale* 倍だけ拡大したものを作成して、それをオペランドスタックにプッシュします。

たとえば、

```
/Times-Roman findfont 24 scalefont
```

というプログラムを実行することによって、`Times-Roman` というフォント名によって識別されるフォント辞書を取得して、それを拡大することによって大きさが 24 ポイントのフォント辞書を作成して、それをオペランドスタックにプッシュすることができます。

3.8.4　グラフィックス状態に対するフォント辞書の設定

グラフィックス状態は、一つのフォント辞書を持っています。カレントグラフィックス状態が持っ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。文字列は、カレントフォントを使って描画されます。

`setfont` というオペレーターを使うことによって、カレントグラフィックス状態に対してフォント辞書を設定することができます。

font `setfont` –　　**オペレーター**

カレントグラフィックス状態に対してフォント辞書 *font* を設定します。

たとえば、

```
/Times-Roman findfont 24 scalefont setfont
```

というプログラムを実行することによって、Times-Roman というフォント名によって識別されるフォント辞書を取得して、それを拡大することによって大きさが 24 ポイントのフォント辞書を作成して、カレントグラフィックス状態に対してそれを設定することができます。

3.8.5 文字列を描画する位置の指定

文字列は、カレントポイントを基準とする位置に描画されます。ですから、カレントページに文字列を描画するためには、それに先立って、カレントポイントの設定をする必要があります。

カレントポイントは、moveto または rmoveto を使うことによって設定することができます（ただし、カレントポイントが存在していない段階では、rmoveto を使うことはできません）。

3.8.6 カレントページへの文字列の描画

show というオペレーターを使うことによって、カレントページに文字列を描画することができます。

string show – **オペレーター**

カレントフォントとカレントカラーを使って、カレントページのカレントポイントに文字列 *string* を描画します。そして、描画した文字列の末尾の位置へカレントポイントを移動させます。

プログラムの例 show.ps

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

3.8.7 文字の間隔

フォントを構成しているそれぞれの文字の形状は、「文字幅」(width of glyph) と呼ばれるベクトルを持っています。show は、文字列を描画するとき、それぞれの文字の位置を文字幅でずらしていきます。

ashow というオペレーターは、show と同じように文字列を描画するのですが、オペランドで指定したベクトルを文字幅に加算した結果で、それぞれの文字の位置をずらしていきます。ですから、ashow を使うことによって、描画する文字列を構成している文字の間隔を広くしたり狭くしたりすることができます。

a_x a_y *string* ashow – **オペレーター**

string を描画します。ただし、(a_x, a_y) というベクトルを文字幅に加算した結果を使って文字の位置をずらします。

プログラムの例 ashow.ps

```
%!PS-Adobe-3.0
/drawalphabet {
    4 dict begin
    /ay exch def
    /ax exch def
    /y exch def
    /x exch def
    0.8 0 0.3 setrgbcolor
    /Times-Roman findfont 18 scalefont setfont
    x y moveto
    ax ay (abcdefghijklmnopqrstuvwxyz) ashow
    end
} def
100 700 0 0 drawalphabet
100 650 4 0 drawalphabet
100 600 8 0 drawalphabet
100 550 -2 0 drawalphabet
100 400 0 4 drawalphabet
```

```
100 350 0 -4 drawalphabet
showpage
```

3.8.8 単語の間隔

widthshow というオペレーターを使うことによって、文字の間隔ではなくて、単語の間隔を広くしたり狭くしたりすることができます。

c_x c_y char string widthshow − **オペレーター**

string を描画します。ただし、整数 *char* を文字コードとする文字については、(c_x, c_y) というベクトルを文字幅に加算した結果を使って、次の文字の位置をずらします。

空白の文字コードは 32 ですので、たとえば、

```
8 0 32 (word spacing) widthshow
```

というプログラムを実行することによって描画される文字列は、 word と spacing とのあいだの空白が、通常よりも 8 だけ広くなります。

プログラムの例 wshow.ps

```
%!PS-Adobe-3.0
/drawwords {
    5 dict begin
    /char exch def
    /ay exch def
    /ax exch def
    /y exch def
    /x exch def
    0.6 0 0.8 setrgbcolor
    /Times-Roman findfont 18 scalefont setfont
    x y moveto
    ax ay char (one two three four five six seven) widthshow
    end
} def
100 700 0 0 32 drawwords
100 650 12 0 32 drawwords
100 600 24 0 32 drawwords
100 550 -2 0 32 drawwords
100 400 0 16 32 drawwords
100 350 0 -16 32 drawwords
100 200 72 0 105 drawwords
showpage
```

3.8.9 欧米語のフォント

欧米語の標準的なフォント辞書としては、次のようなフォント名を持つものがあります。

```
Times-Roman  Helvetica  Courier  Symbol
```

プログラムの例 font.ps

```
%!PS-Adobe-3.0
/drawfontname {
    3 dict begin
    /fontname exch def
    /y exch def
    /x exch def
    0.4 0.6 0 setrgbcolor
    fontname cvn findfont 64 scalefont setfont
    x y moveto
    fontname show
    end
} def
100 600 (Times-Roman) drawfontname
100 500 (Helvetica) drawfontname
100 400 (Courier) drawfontname
100 300 (Symbol) drawfontname
showpage
```

3.8.10 日本語のフォント

日本語の標準的なフォント辞書としては、次のようなフォント名を持つものがあります。

`Ryumin-Light-RKSJ-H`	明朝体の横組み。
`Ryumin-Light-RKSJ-V`	明朝体の縦組み。
`GothicBBB-Medium-RKSJ-H`	ゴシック体の横組み。
`GothicBBB-Medium-RKSJ-V`	ゴシック体の縦組み。

ちなみに、これらのフォント名を持つフォント辞書は、文字コードとして Shift_JIS を使っています。これらのフォント名に含まれている `RKSJ` という部分を `EUC` に置き換えると、文字コードとして EUC-JP を使うフォント辞書の名前になります。

日本語の文字列を PostScript のプログラムの中に書く場合、

(日本語の文字列)

というように、それを丸括弧の中にそのまま書いたとすると、そのプログラムが 7 ビットの伝送路を通過したときに、文字列が文字化けしてしまいます。ですから、日本語の文字列を PostScript のプログラムの中に書く場合は、通常、

```
<93fa 967b 8cea 82cc 95b6 8e9a 97f1>
```

というように、小なりと大なりで 16 進数の列を囲んだ形式でそれを記述します。

プログラムの例 `nihongo.ps`

```
%!PS-Adobe-3.0
/drawnihongo {
    3 dict begin
    /fontname exch def
    /y exch def
    /x exch def
    0 0.4 0.8 setrgbcolor
    fontname findfont 64 scalefont setfont
    x y moveto
    <90bc 9286 9387 93ec 95fb> show
    end
} def
100 650 /Ryumin-Light-RKSJ-H drawnihongo
100 550 /GothicBBB-Medium-RKSJ-H drawnihongo
200 450 /Ryumin-Light-RKSJ-V drawnihongo
300 450 /GothicBBB-Medium-RKSJ-V drawnihongo
showpage
```

3.8.11 文字列の配置

何個かの文字列を並べて描画するときに、文字列のどの位置を基準として並べるかということを、文字列の「配置」(alignment) と言います。

横組みの文字列を並べる場合には、通常、次のいずれかの配置が使われます。

左寄せ (flush left)	文字列の左端を基準とする配置。
右寄せ (flush right)	文字列の右端を基準とする配置。
センタリング (centering)	文字列の左右方向の中央を基準とする配置。

`show` などを使って文字列を描画する場合、オペランドで指定した x 座標が文字列の左端の位置になりますから、左寄せで文字列を並べるというのは、簡単に実現することができます。それに対して、右寄せまたはセンタリングで文字列を並べるためには、形状としての文字列の長さを求める必要があります。

形状としての文字列の長さは、`stringwidth` というオペレーターを使うことによって求めることができます。

string `stringwidth` w_x w_y **オペレーター**

カレントフォントを使って文字列 *string* を描画したときにカレントポイントが移動する距離を求めます。w_x が x 軸方向の移動距離、w_y が y 軸方向の移動距離です。

プログラムの例 align.ps

```
%!PS-Adobe-3.0
/flushleft {
    3 dict begin
    /string exch def
    /y exch def
    /x exch def
    x y moveto
    string show
    end
} def
/flushright {
    3 dict begin
    /string exch def
    /y exch def
    /x exch def
    x y moveto
    string stringwidth pop neg 0 rmoveto
    string show
    end
} def
/centering {
    3 dict begin
    /string exch def
    /y exch def
    /x exch def
    x y moveto
    string stringwidth pop neg 2 div 0 rmoveto
    string show
    end
} def
/threestrings {
    4 dict begin
    /proc exch def
    /dy exch def
    /y exch def
    /x exch def
    x y moveto
    x y (hotokenoza) proc
    x y dy sub (renge) proc
    x y dy 2 mul sub (nanohana) proc
    end
} def
/Times-Roman findfont 24 scalefont setfont
0 0.6 0.2 setrgbcolor
100 500 32 { flushleft } threestrings
500 500 32 { flushright } threestrings
300 500 32 { centering } threestrings
showpage
```

3.8.12 文字列の輪郭

文字列を描画したいときは、基本的には show などのオペレーターを使えばいいわけですが、この方法では、文字列という形状の内部を塗りつぶすことだけしかできません。

文字列は、その形状の内部を塗りつぶすことだけではなくて、その形状を構成している線を描画するということ、つまり文字列の輪郭だけを描画するということも可能です。ただし、そのためには、文字列の形状をパスとしてカレントパスに追加する、という操作をする必要があります。

文字列の形状は、 charpath というオペレーターを使うことによって、カレントパスに追加することができます。

string bool charpath – **オペレーター**

カレントフォントを使ってカレントポイントに文字列 *string* を描画するためのパスを作成して、それをカレントパスに追加します。そして、文字列の末尾の位置へカレントポイントを移動させます。

bool は、塗りつぶすのに適したパスを作成するか、それとも線を描画するのに適したパスを作

成するか、ということを指示する真偽値です。真は前者を指示して、偽は後者を指示します。

プログラムの例 charpath.ps

```
%!PS-Adobe-3.0
newpath
/Times-Roman findfont 230 scalefont setfont
0 0.6 0.8 setrgbcolor
3 setlinewidth
100 400 moveto
(char) false charpath
stroke
/Ryumin-Light-RKSJ-V findfont 320 scalefont setfont
300 750 moveto
<95b6 8e9a> false charpath
0.8 0 0.6 setrgbcolor
4 setlinewidth
stroke
showpage
```

3.9 座標系の変換

3.9.1 座標系の変換の基礎

第 3.1 節で説明したように、グラフィックス状態は、原点の位置、軸の方向、長さの単位という、座標系に関するデータを持っています。そして、デフォルトの座標系では、原点の位置はカレントページの左下の隅、x 軸の方向は右、y 軸の方向は上、長さの単位は 1 ポイントになっています。

グラフィックス状態が持っている座標系に関するデータは、オペレーターを実行することによって、変更することが可能です。それらを変更するという操作は、座標系の「変換」(transformation) と呼ばれます。

3.9.2 移動

座標系の原点の位置を変更するという操作は、座標系の「移動」(translation) と呼ばれます。

座標系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。

t_x t_y translate – **オペレーター**

カレントグラフィックス状態が持っている原点の位置を、x 軸方向へ t_x、y 軸方向へ t_y だけ移動させます。

プログラムの例 transla.ps

```
%!PS-Adobe-3.0
/hline {
    newpath
    0 0 moveto
    200 0 rlineto
    stroke
} def
/translatehline {
    100 100 translate
    22 {
        hline
        8 30 translate
    } repeat
} def
0.3 0.8 0 setrgbcolor
16 setlinewidth
translatehline
showpage
```

3.9.3 回転

座標系の軸の方向を変更するという操作は、座標系の「回転」(rotation) と呼ばれます。

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。

angle rotate – **オペレーター**

カレントグラフィックス状態が持っている座標系の軸の方向を、反時計回りに *angle* 度だけ回転させます。

プログラムの例 rotate.ps

```
%!PS-Adobe-3.0
/hline {
    newpath
    60 0 moveto
    160 0 rlineto
    stroke
} def
/rotatehline {
    300 450 translate
    36 {
        hline
        10 rotate
    } repeat
} def
0 0.6 0.8 setrgbcolor
6 setlinewidth
rotatehline
showpage
```

3.9.4 拡大

長さの単位を変更するという操作は、座標系の「拡大」(scaling) と呼ばれます。変更前の単位に対する変更後の単位の比率、つまり単位を何倍するかという数値は、「拡大率」(scaling factor) と呼ばれます。

長さの単位は、x 軸方向と y 軸方向のそれぞれを、異なる拡大率で拡大することができます。

デフォルトの座標系では、1 という長さは 1 ポイントを意味しているわけですが、たとえば x 軸方向に 2 倍に拡大された座標系では、x 軸方向の 1 という長さは 2 ポイントを意味することになります。

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。

s_x s_y scale – **オペレーター**

カレントグラフィックス状態が持っている長さの単位を、x 軸方向に s_x 倍、y 軸方向に s_y 倍だけ拡大します。

プログラムの例 scale.ps

```
%!PS-Adobe-3.0
/hline {
    newpath
    0 0 moveto
    100 0 rlineto
    stroke
} def
/scalehline {
    100 150 translate
    10 {
        hline
        1.16 0.82 scale
        0 150 translate
    } repeat
} def
0.8 0 0.6 setrgbcolor
100 setlinewidth
scalehline
showpage
```

3.10 クリッピング

3.10.1 クリッピングとは何か

PostScript のプログラムは、「クリッピング」(clipping) という処理を実行することができます。

何らかの図形に沿ってグラフィックスを切り抜くことを、そのグラフィックスを「クリップする」(clip) と言います。「クリッピング」というのは、この動詞の名詞形です。

PostScript のプログラムが実行することのできるクリッピングというのは、グラフィックスを描画することのできる領域を、何らかの図形の内部に制限しておいて、その制限のもとでグラフィックスを描画するということです。このような処理を実行することによって、何らかの模様を持つ図形を描画することができます。

3.10.2 クリッピングパス

グラフィックス状態は、「クリッピングパス」(clipping path) と呼ばれるパスを持っています。カレントグラフィックス状態が持っているクリッピングパスは、「カレントクリッピングパス」(current clipping path) と呼ばれます。

カレントページに対するグラフィックスの描画は、カレントクリッピングパスで囲まれた領域だけに制限されます。ですから、何らかのパスをカレントクリッピングパスとして設定して、そののちにグラフィックスを描画することによって、そのパスを使ってグラフィックスをクリップすることができます。

クリッピングパスは、`clip` または `eoclip` というオペレーターを使うことによって設定することができます。

`clip` と `eoclip` は、カレントクリッピングパスをさらに狭い領域へ制限するという動作をします。ですから、カレントクリッピングパスとは無関係に、新しいパスをカレントクリッピングパスとして設定したい場合は、カレントクリッピングパスを初期状態に戻す必要があります。カレントクリッピングパスは、`initclip` というオペレーターを使うことによって、初期状態に戻すことができます。

— clip — **オペレーター**

カレントパスをカレントクリッピングパスとして設定します。領域の内部性はワインディング規則によって判定します。

— eoclip — **オペレーター**

カレントパスをカレントクリッピングパスとして設定します。領域の内部性は偶奇規則によって判定します。

— initclip — **オペレーター**

カレントクリッピングパスを初期状態に戻します。

プログラムの例 `clip.ps`

```
%!PS-Adobe-3.0
/star {
    2 dict begin
    /y exch def
    /x exch def
    newpath
    x y moveto
    300 0 rlineto
    -250 -200 rlineto
    100 300 rlineto
    100 -300 rlineto
    end
} def
/polkadot {
    4 dict begin
    /y exch def
    /x exch def
    /dy y def
    14 {
        /dy dy 25 add def
        /dx x def
        14 {
```

```
            /dx dx 25 add def
            newpath
            dx dy 8 0 360 arc
            stroke
        } repeat
    } repeat
    end
} def
/polkadotstar {
    150 650 star
    clip
    100 400 polkadot
} def
/polkadotstarbyeo {
    150 300 star
    eoclip
    100 50 polkadot
} def
1 0 0.4 setrgbcolor
6 setlinewidth
polkadotstar
initclip
polkadotstarbyeo
showpage
```

3.11　グラフィックス状態の保存

3.11.1　グラフィックス状態を構成しているデータ

第 3.1 節で説明したように、グラフィックス状態というのは、グラフィックスに関連するさまざまな状態をあらわしているデータの集合のことです。

グラフィックス状態を構成しているデータのうちで、これまでに登場したものとしては、次のようなものがあります。

> パス、ペン位置、線幅、色、端点の形状、接続点の形状、マイターリミット、破線配列、破線オフセット、フォント辞書、原点の位置、軸の方向、長さの単位、クリッピングパス

3.11.2　グラフィックス状態スタック

PostScript のインタプリタは、「グラフィックス状態スタック」(graphics state stack) と呼ばれるスタックを持っています。

グラフィックス状態スタックの中には、グラフィックス状態のコピーを保存しておくことができます。ですから、

(1) グラフィックス状態のコピーを保存する。
(2) グラフィックス状態を変更してグラフィックスを描画する。
(3) 保存しておいたグラフィックス状態を復元する。

という手順を実行することによって、グラフィックス状態に対する変更を局所的なものにして、その影響が他の部分へ波及することを防止することができます。

グラフィックス状態は、`gsave` というオペレーターを使うことによって、そのコピーを保存しておくことができます。そして、保存しておいたグラフィックス状態は、`grestore` というオペレーターを使うことによって復元することができます。

— `gsave` — **オペレーター**

カレントグラフィックス状態のコピーをグラフィックス状態スタックにプッシュします。

— `grestore` — **オペレーター**

グラフィックス状態スタックからグラフィックス状態をポップして、そのグラフィックス状態をカレントグラフィックス状態にします。

プログラムの例　`gsave.ps`

```
%!PS-Adobe-3.0
/square {
```

```
    2 dict begin
    /y exch def
    /x exch def
    x y moveto
    200 0 rlineto
    0 200 rlineto
    -200 0 rlineto
    fill
    end
} def
/specialsquare {
    gsave
    200 200 translate
    50 rotate
    2 0.5 scale
    0.4 0.8 0 setrgbcolor
    0 0 square
    grestore
} def
0 0.4 0.8 setrgbcolor
200 100 square
specialsquare
200 500 square
showpage
```

このプログラムの中で定義されている specialsquare という手続きは、最初にグラフィックス状態のコピーを保存して、最後にそれを復元しています。ですから、この手続きの中でグラフィックス状態に対して加えられた変更は、この手続きの外側にはまったく影響を及ぼしません。

3.11.3 パスの再利用

カレントパスを描画するオペレーター、つまり、stroke、fill、eofill というオペレーターは、カレントパスを描画したのち、それを初期化します。ですから、それらのオペレーターを実行した直後は、カレントパスが空の状態になっています。

しかし、パスを描画したのち、そのパスを再利用したい、ということもしばしばあります。たとえば、一つのパスについて、領域の塗りつぶしと線の描画を両方とも実行したい、というような場合です。

パスは、グラフィックス状態のコピーを保存しておくことによって再利用することができます。つまり、パスを構築したのちに gsave を実行して、そのパスを描画したのちに grestore を実行すれば、そのパスをふたたび描画することができるわけです。

プログラムの例 filstro.ps

```
%!PS-Adobe-3.0
0 0.4 0 setrgbcolor
30 setlinewidth
newpath
100 600 moveto
400 0 rlineto
0 -300 rlineto
-400 0 rlineto
closepath
gsave
1 1 0.4 setrgbcolor
fill
grestore
stroke
showpage
```

このプログラムは、 gsave と grestore を使うことによって、一つの長方形のパスについて、その領域の塗りつぶしと線の描画を両方とも実行します。

3.12　DSC

3.12.1　DSC とは何か

PostScript で書かれたプログラムを処理するプログラムは、「文書マネージャー」(document manager) と呼ばれます。

PostScript という言語は、「DSC」と呼ばれる規約を含んでいます。この規約にしたがって PostScript のプログラムを書くことによって、そのプログラムに関するさまざまな情報を文書マネージャーに伝えることができます。ちなみに、DSC というのは、「文書構造化規約」(Document Structuring Conventions) という言葉の略称です。

3.12.2　DSC コメント

DSC という規約が定めているのは、「DSC コメント」(DSC comment) と呼ばれる注釈の構文と意味です。文書マネージャーは、PostScript のプログラムの中に書き込まれた DSC コメントから、そのプログラムに関する各種の情報を読み取ります。

1 個の DSC コメントは、1 行の文字列です。その先頭には、通常、2 個のパーセント (`%%`) を書きます。そして、その右側に、「コメントキーワード」(comment keyword) と呼ばれる、何を記述している DSC コメントなのかということを示す名前を書きます。たとえば、PostScript のプログラムの末尾を明示的に示す DSC コメントは、`EOF` というコメントキーワードを使って、

```
%%EOF
```

と書きます。

3.12.3　DSC コメントの引数

コメントキーワードには、その末尾にコロン (`:`) があるものとないものとがあります。コメントキーワードの末尾にあるコロンは、そのコメントキーワードを使って DSC コメントを書く場合、コメントキーワードの右側にさらに何かを書く必要がある、ということを意味しています。

DSC コメントの一部分で、コメントキーワードの右側に書かれているものは、その DSC コメントの「引数」(argument) と呼ばれます。

たとえば、PostScript のプログラムに対してタイトルを与える DSC コメントを書く場合に使われるコメントキーワードは、`Title:` です。このコメントキーワードの末尾にはコロンがありますので、それを使う DSC コメントには、引数を書く必要があります。

`Title:` を使って書かれた DSC コメントは、その DSC コメントを含んでいるプログラムに対して、その引数をタイトルとして与える、ということを意味しています。たとえば、

```
%%Title: (I Am a Title of This Program)
```

という DSC コメントは、この DSC コメントを含んでいるプログラムに対して、

```
I Am a Title of This Program
```

という文字列をタイトルとして与える、ということを意味しています。

プログラムにタイトルを与える DSC コメントを書く場合、その引数は、丸括弧で囲んでも囲まなくても、どちらでもかまいません。

3.12.4　プログラムの先頭に書く DSC コメント

第 1.2 節で説明したように、PostScript のプログラムをファイルに保存する場合には、その先頭の行に、

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

という記述を書くことになっています。実は、この記述も DSC コメントです（この DSC コメントの先頭にはパーセント (`%`) が 1 個だけしかありませんが、これはあくまで例外です)。

この DSC コメントは、その下に書かれているのが PostScript のプログラムだということと、使われている DSC のバージョンが 3.0 だということを示しています。

3.12.5　複数のページを扱うプログラム

PostScript で書かれた 1 個のプログラムは、複数のページのそれぞれにグラフィックスを描画することができます。ただし、複数のページを扱うプログラムを書くためには、その中に 2 種類の DSC コメントを書く必要があります。

第一に必要なのは、全体のページ数を記述する DSC コメントです。全体のページ数を記述する DSC コメントは、Pages: というコメントキーワードを使って、

%%Pages: ページ数

と書きます。「ページ数」のところには、全体のページ数を示すプラスの整数を書きます。たとえば、

```
%%Pages: 76
```

という DSC コメントは、この DSC コメントを含んでいるプログラムが扱うページが、全体で 76 ページあるということを意味しています。

第二に必要なのは、それぞれのページにグラフィックスを描画する記述とページの番号とを対応させる DSC コメントです。そのような DSC コメントは、Page: というコメントキーワードを使って、

%%Page: ラベル 番号

と書きます。「ラベル」のところには任意の文字列、「番号」のところにはプラスの整数を書きます。この DSC コメントは、その下に書かれている記述が、「番号」のところに書かれたプラスの整数を番号とするページにグラフィックスを描画するものだということ（番号は、0 からではなくて 1 から始まります）、そして、「ラベル」のところに書かれた文字列を、そのページに対してラベルとして与える、ということを意味しています。たとえば、

```
%%Page: (title page for Chapter Five) 63
```

という DSC コメントは、その下に書かれている記述が、63 ページ目にグラフィックスを描画するものだということ、そして、そのページに対して、

```
title page for Chapter Five
```

という文字列をラベルとして与える、ということを意味しています。

ラベルとしてページに与える文字列が、空白もエスケープシーケンスも含んでいない場合、その文字列を囲む丸括弧は、省略することができます。たとえば、26 ページ目に対して xxvi というラベルを与える DSC コメントは、

```
%%Page: xxvi 26
```

というように、丸括弧を省略して書いてもかまいません。

複数のページを扱うプログラムを書く場合は、それぞれのページごとに showpage が実行されるようにする必要があります。

プログラムの例 page.ps

```
%!PS-Adobe-3.0
%%Pages: 3
/showstring {
    1 dict begin
    /string exch def
    100 500 moveto
    0 0.6 0 setrgbcolor
    /Times-Roman findfont 128 scalefont setfont
    string show
    showpage
    end
} def
%%Page: (first page) 1
(first) showstring
%%Page: (second page) 2
(second) showstring
%%Page: (third page) 3
(third) showstring
```

このプログラムは、1 ページ目に first、2 ページ目に second、3 ページ目に third という文字列を描画します。

3.13　EPS

3.13.1　EPS とは何か

グラフィックスのデータ形式のひとつに、「EPS」と呼ばれるものがあります。この形式で作られたデータは、 .eps という拡張子を持つファイルに保存されます。

グラフィックスを扱うプログラムの多くは、グラフィックスを EPS で保存したり、EPS のデータを読み込んで、それを別のグラフィックスに貼り付けたりすることができます。

EPS というのは、「カプセル化された PostScript」(Encapsulated PostScript) という言葉の略称です。この言葉からも分かるとおり、EPS という形式のデータは、PostScript を使ってグラフィックスを記述したものです。

「カプセル化された」(encapsulated) というのは、「周囲に影響を及ぼさない」という意味です。EPS のデータは、PostScript のプログラムの一部分としてそれを取り込んだ場合に、その周囲に影響を及ぼさない、という特徴を持っています。

3.13.2　EPS の書き方

EPS のデータというのは PostScript のプログラムなのですが、その書き方は、PostScript の普通のプログラムの書き方とは少し違っています。EPS のデータは、次の規則を守って書かないといけないということになっています。

- 先頭の行に、次の DSC コメントを書かないといけません。

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

- バウンディングボックスコメントを書かないといけません。
- 複数のページにグラフィックスを描画してはいけません。
- インタプリタの状態を変更するような特殊なオペレーターを実行することはできません。

バウンディングボックスコメントについては、次の項で説明したいと思います。

EPS のデータは、自分自身で showpage を実行する必要はありません。とは言っても、絶対に実行してはいけないというわけでもありません。EPS のデータは、 showpage が、

```
/showpage { } def
```

と再定義された状態で実行されますので、EPS のデータが showpage を実行したとしても、問題は何も生じません。

3.13.3　バウンディングボックスコメント

何らかのグラフィックスについて、それを取り囲む長方形の領域のうちで最小のものは、そのグラフィックスの「バウンディングボックス」(bounding box) と呼ばれます。

PostScript のプログラムの中には、そのプログラムが描画するグラフィックスのバウンディングボックスの位置と大きさを示す DSC コメントを書くことができます。そのような DSC コメントは、「バウンディングボックスコメント」(bounding box comment) と呼ばれます。

バウンディングボックスコメントは、 BoundingBox: というコメントキーワードを使って、

```
%%BoundingBox: llx lly urx ury
```

と書きます。*llx*、*lly*、*urx*、*ury* のそれぞれの場所には、1 個の整数を書きます。そうすると、(llx, lly) がバウンディングボックスの左下の頂点の座標、(urx, ury) が右上の頂点の座標を意味することになります。たとえば、

```
%%BoundingBox: 160 140 380 570
```

という DSC コメントは、この DSC コメントを含んでいる PostScript のプログラムが描画するグラフィックスのバウンディングボックスが、左下の頂点の座標が $(160, 140)$ で、右上の頂点の座標が $(380, 570)$ であるような長方形だということを意味しています。

EPS のデータを書く場合には、そのデータの中に、そのデータが描画するグラフィックスのバウンディングボックスの位置と大きさを示す DSC コメントを書く必要があります。

EPS のデータの例　circle.eps

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 160 360 440 640
newpath
```

```
300 500 100 0 360 arc
closepath
0 0.6 0.8 setrgbcolor
80 setlinewidth
stroke
```

参考文献

[EPSFFS,1992] Adobe Systems Incorporated, “Encapsulated PostScript File Format Specification, Version 3.0”, Adobe Technical Note #5002, 1992.

[PSLDSCS,1992] Adobe Systems Incorporated, “PostScript Language Document Structuring Conventions Specification, Version 3.0”, Adobe Technical Note #5001, 1992.

[PSLPD,1988] Adobe Systems Incorporated, *PostScript Language Program Design*, Addison-Wesley, 1988, ISBN 978-0-201-14396-6. 邦訳（松村邦仁、アスキー出版技術部）、『ページ記述言語 PostScript プログラム・デザイン』、アスキー、1990、ISBN 978-4-7561-0047-4。

[PSLRM,1999] Adobe Systems Incorporated, *PostScript Language Reference Manual, Third Edition*, Addison-Wesley, 1999, ISBN 978-0-201-37922-8. 邦訳（桑沢清志）、『PostScript リファレンスマニュアル・第 3 版』、アスキー、2001、ISBN 978-4-7561-3822-4。

[PSLTC,1985] Adobe Systems Incorporated, *PostScript Language Tutorial and Cookbook*, Addison-Wesley, 1985, ISBN 978-0-201-10179-9. 邦訳（野中浩一）、『PostScript チュートリアル&クックブック』、アスキー、1989、ISBN 978-4-7561-0005-4。

[安居院,1993] 安居院猛、永江孝規、『PostScript グラフィクス』、新紀元社、1993、ISBN 978-4-88317-032-6。

[江口,1997] 江口庄英、『Ghostscript Another Manual』、ソフトバンク、1997、ISBN 978-4-7973-0344-5。

[森,1997] 森茂樹、渋谷壮一、『PostScript 詳細解説』、CQ 出版、1997、ISBN 978-4-7898-1851-3。

索引