Introdução a PostScript

Hudson Lacerda (2004)

O objetivo do presente texto é apenas iniciar o leitor ao uso de *PostScript*. A linguagem não é descrita formalmente, e somente alguns poucos operadores são apresentados. No entanto, espera−se que este texto seja útil como ponto de partida e estímulo ao uso e estudo de *PostScript*.

Para informações precisas e detalhadas sobre *PostScript*, consultem−se os textos oficiais disponíveis no sítio da Adobe (em especial os livros `PLRM.pdf` e `bluebook.pdf`):

`http://www.adobe.com/`

PostScript é uma linguagem de descrição de gráficos criada por Adobe Systems Inc., para controle de impressoras. Sua especificação é aberta, então existem diversos programas (e impressoras) capazes de lidar com *PostScript*.

Um arquivo *PostScript* é um arquivo de texto contendo comandos para a geração de uma ou mais páginas. O arquivo deve começar com os caracteres **`'%!'`**, que indicam ao programa interpretador que o arquivo contém comandos *PostScript*. Ao final de cada página gerada, *deve−se* utilizar o operador **`showpage`**, que 'imprime' a página atual e prepara o interpretador para uma nova página.

A linguagem *PostScript* baseia−se em uma pilha de dados e comandos. Todo valor escrito no código é armazenado na pilha se for um operando, mas se esse valor é um operador (comando), ele é interpretado, possivelmente modificando os dados da pilha.

Assim, todos os operadores em *PostScript* são pós−fixados, isto é, são escritos *após* seus argumentos.

Para somar os dois valores 3 e 2, o código é o seguinte:

```
3 2 sum
```

Isso coloca os valores 3 e 2 na pilha. Quando o comando **`sum`** é encontrado, ele é executado, tomando os dois valores do topo da pilha e retornando sua soma. Como resultado, após a execução do código acima, a pilha conterá o valor 5 (a soma de 3 e 2). Ao longo desse tutorial, a execução de um trecho de código *PostScript* (e seu efeito sobre a pilha de operandos) será representada como no exemplo abaixo:

```
3 2 sum   =>   5
```

A sintaxe dos operadores é representada da seguinte maneira:

```
argumento(s)  operador  resultado(s)
```

A ausência de argumentos ou resultados será representada por um **`'-'`** na posição correspondente, tal como em:

```
-  currentpoint  x y
```

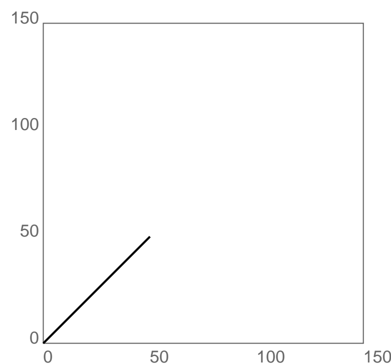
Isso indica que o operador **`currentpoint`** não toma nenhum argumento de entrada (−) e retorna dois valores (x e y) como resultado.

Desenhando gráficos

Em *PostScript*, gráficos são desenhados num plano cartesiano. Mais precisamente, no primeiro quadrante de um plano cartesiano, cujo ponto (0, 0) corresponde à extremidade inferior esquerda da página. A unidade de medida é o *ponto* (pt), que equivale a 0.1389pol (1/72 de polegada) ou 0.3527cm (1cm = 28.35pt).

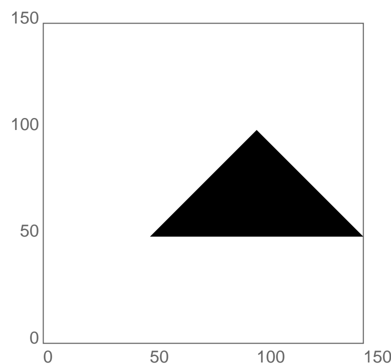
Um gráfico é realizado em duas etapas: inicialmente, ele é 'projetado' utilizando−se operadores como **`moveto`** (mover para o ponto) e **`lineto`** (fazer uma linha até o ponto); finalmente, realizado através de um dos operadores **`stroke`** (traçar) ou **`fill`** (preencher).

Desenhando linhas e polígonos



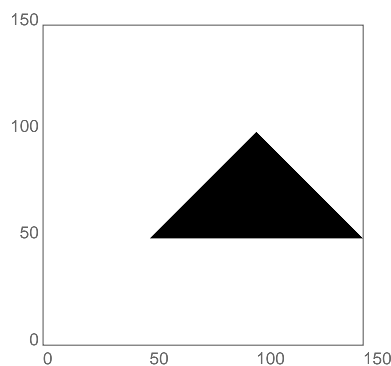
O exemplo a seguir move para o ponto (0, 0), então ‘projeta’ uma linha do ponto atual até o ponto (50, 50), e por fim, traça a linha.

```
0 0 moveto
50 50 lineto
stroke
```



O próximo exemplo ilustra o uso do operador **`fill`**. Um triângulo é ‘projetado’ com os comandos **`moveto`** e **`lineto`**, e então preenchido com **`fill`**.

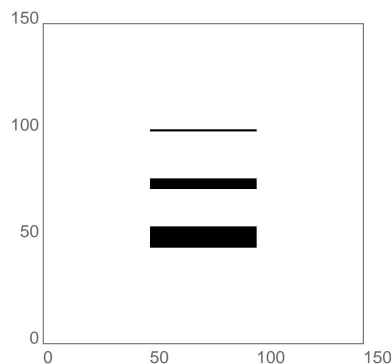
```
50 50 moveto
100 100 lineto
150 50 lineto
50 50 lineto
fill
```



Desde que tenha sido usado o operador **`moveto`** para definir o ponto inicial em um gráfico, é possível empregar os operadores **`rmoveto`** e **`rlineto`** em lugar de **`moveto`** e **`lineto`**. Esses operadores utilizam como parâmetros os *deslocamentos* nos eixos x e y, em relação ao ponto atual.

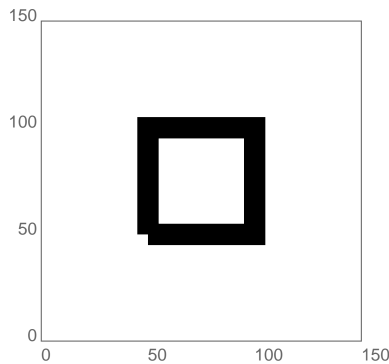
O mesmo triângulo do exemplo anterior pode ser escrito:

```
50 50 moveto
50 50 rlineto
50 -50 rlineto
-100 0 rlineto
fill
```



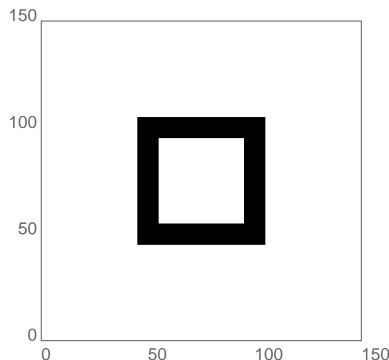
A espessura da linha desenhada pelo operador **`stroke`** pode ser determinada com **`setlinewidth`**. Esse operador utiliza um valor em pontos como argumento. O valor *default* é 1pt.

```
1 setlinewidth
50 100 moveto
50 0 rlineto stroke

5 setlinewidth
50 75 moveto
50 0 rlineto stroke

10 setlinewidth
50 50 moveto
50 0 rlineto stroke
```



O exemplo seguinte tenta traçar um quadrado com linhas de espessura 10pt. Note que não ocorre a junção das linhas no ponto inicial/final (50, 50).

```
50 50 moveto
0 50 rlineto
50 0 rlineto
0 -50 rlineto
-50 0 rlineto
10 setlinewidth
stroke
```



O quadrado pode ser melhorado com o uso do comando **closepath**. Ele desenha uma linha do ponto atual até o ponto inicial (definido pelo último comando **moveto**), realizando a junção desejada.

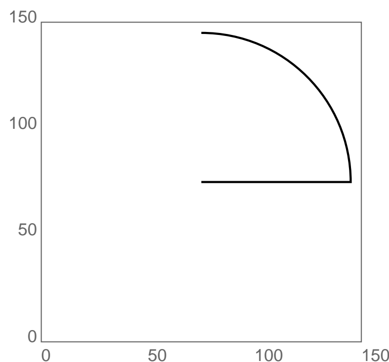
```
50 50 moveto
0 50 rlineto
50 0 rlineto
0 -50 rlineto
closepath
10 setlinewidth
stroke
```

Desenhando arcos e círculos

Os operadores **arc** e **arcn** permitem o desenho de circunferências, círculos, arcos e formas derivadas. Esses operadores utilizam cinco argumentos: as coordenadas x e y do centro da circunferência, o raio, o ângulo inicial e o ângulo final:

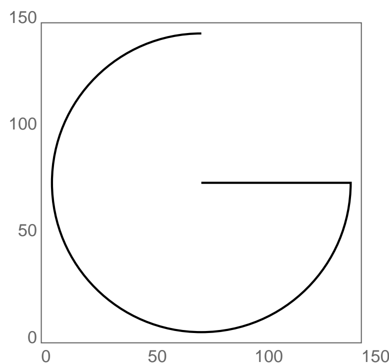
```
xc yc r ang1 ang2  arc  -
xc yc r ang1 ang2  arcn  -
```

Se existir um 'ponto atual', será desenhada uma linha desse ponto até o início do arco.



arc desenha um arco em sentido anti–horário.

```
75 75 moveto
75 75
70
0 90
arc
stroke
```

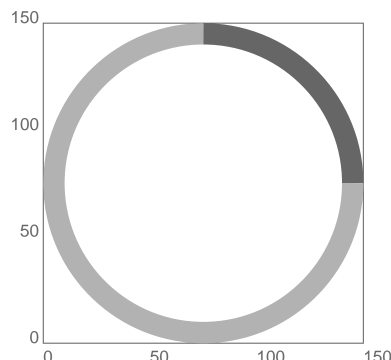


arcn desenha um arco em sentido horário.

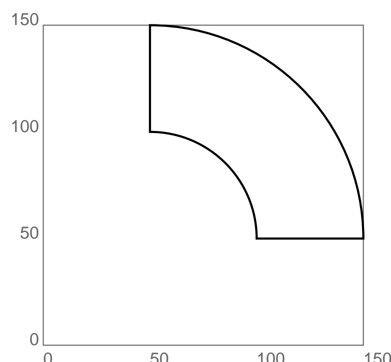
```
75 75 moveto
75 75
70
0 90
arcn
stroke
```

Pode−se garantir que a linha não será desenhada precedendo−se o comando **`newpath`** ao código do arco. Os comandos **`fill`** e **`stroke`** realizam **`newpath`** implicitamente ao final de sua execução: como resultado, o 'ponto atual' é indefinido.

O próximo exemplo também ilustra o operador **`setgray`**, que determina o 'tom de cinza' a utilizar no gráfico. O argumento deve ser um valor de 0.0 (preto − o *default*) a 1.0 (branco).



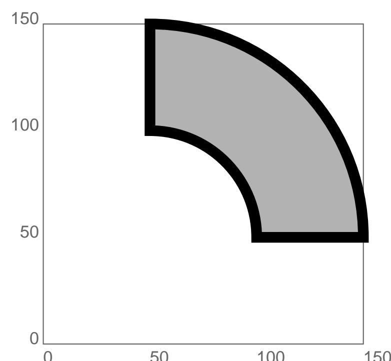

```
newpath
10 setlinewidth
.4 setgray
75 75 70 0 90 arc
stroke
.7 setgray
75 75 70 0 90 arcn
stroke
```



Nesse exemplo, **`arc`** gera um arco iniciado no ponto (150,50) e terminado em (50,150). **`arcn`** une esse ponto ao ponto (50,100), e então gera um arco até (100,50). Então, **`closepath`** liga esse ponto ao ponto inicial (150,50). Finalmente, o gráfico é traçado pelo operador **`stroke`**.

```
newpath
50 50 100 0 90 arc
50 50 50 90 0 arcn
closepath
stroke
```

Um gráfico − incluindo diversas informações como cor e espessura de linha − pode ser salvo para reutilização, através do par de comandos **`gsave`** e **`grestore`**.



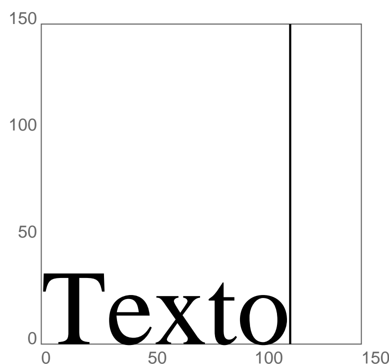
O gráfico do exemplo anterior pode ser preenchido (**`fill`**) e contornado (**`stroke`**), sem haver necessidade de escrever o código duas vezes.

```
newpath
50 50 100 0 90 arc
50 50 50 90 0 arcn
closepath
gsave
   .7 setgray fill
grestore
5 setlinewidth stroke
```

Os operadores **`gsave`** e **`grestore`** são também utilizados para isolar trechos de código, evitando que certas operações (como por exemplo mudança de cor) contaminem os gráficos subseqüentes. Note no exemplo acima que o comando **`.7 setgray`**, delimitado por **`gsave`** e **`grestore`**, não faz com que o contorno (**`stroke`**) seja realizado em cinza, porque **`grestore`** restaura as propriedades gráficas anteriores a **`gsave`**.

Texto

Para se inserir texto em *PostScript*, primeiro seleciona−se uma fonte de caracteres (que deve ser uma fonte *PostScript*, e não uma fonte do sistema operacional), define−se o tamanho, move−se para o ponto onde o texto deve ser inserido e então imprime−se o texto.



Note–se que um nome literal (ou seja, um nome que não deve ser interpretado) é iniciado com o caracter **'/'**. Uma *string* (isto é, uma seqüência de caracteres) é delimitada por parênteses. O operador **show** imprime a *string* dada como argumento a partir do ponto atual; ao final de sua execução, o ponto atual estará pouco além do último caracter impresso.

```
/Times-Roman findfont
50 scalefont
setfont
0 0 moveto
(Texto) show
0 150 rlineto stroke
```

Alguns operadores de *strings* e caracteres:

`n` **`string`** `string`
Cria uma *string* com n elementos. O exemplo abaixo cria uma *string* associada ao nome **texto**;
```
/texto 20 string def
```

`string` **`length`** `n`
Número de elementos da *string*.
```
(texto) length => 5
```

`string bool` **`charpath`** `-`
Cria o contorno dos caracteres da *string*. Exemplo de uso:
```
(vazado) true charpath stroke
```

vazado

`string` **`stringwidth`** `wx wy`
Largura da *string* na fonte atual. Exemplo:
```
/Helvetica findfont 50 scalefont setfont
(teste) stringwidth => 108.4 0.0
```

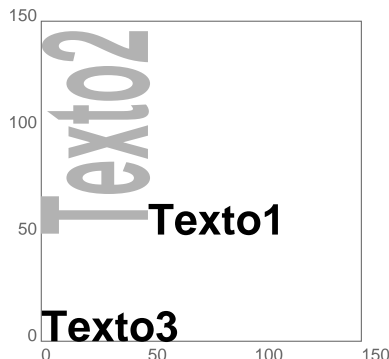
Transformações do sistema de coordenadas

O sistema de coordenadas pode ser manipulado através de certos operadores, tais como:

```
fx fy scale -
x y translate -
ang rotate -
```

scale amplia/encolhe o sistema de coordenadas atual segundo os fatores dados. **translate** transporta o sistema de modo que ponto especificado se torna a nova origem (0, 0). **rotate** realiza uma rotação em sentido anti–horário, segundo o ângulo especificado.

É altamente recomendável delimitar o código que utilizar algum desses operadores com **gsave** e **grestore**.




```
/Helvetica-Bold findfont 20 scalefont setfont
gsave
   50 50 translate
   0 0 moveto (Texto1) show
   90 rotate
   1.5 3.5 scale
   .7 setgray
   0 0 moveto (Texto2) show
grestore
0 0 moveto (Texto3) show
```

Definindo novos operadores

Novos operadores são definidos com **def**. A sintaxe é a seguinte:

`nome valor` **def** `-`

O `nome` é escrito iniciando com **'/'** (nome literal), e o `valor` pode ser de qualquer tipo (número, *string*, *array*, procedimento, literal, booleano).

Por exemplo, para definir um operador que calcula a média aritmética de dois números:

```
/media { sum 2 div } def
```

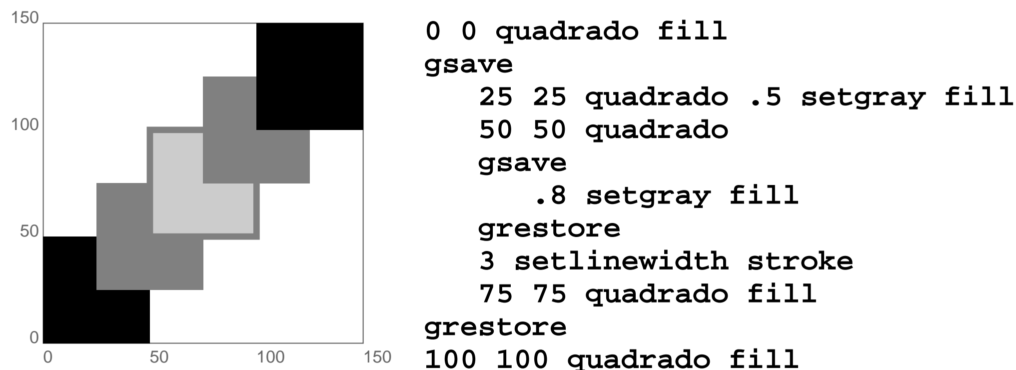
Nesse caso, o `valor` é um procedimento (delimitado por chaves).

O exemplo seguinte define o operador **quadrado**, que 'planeja' um quadrado de lado 50pt cujo canto inferior esquerdo situa–se no ponto definido pelos dois argumentos esperados.

O operador **bind** substitui os nomes de operadores no procedimento pelos respectivos operadores, otimizando a execução.

```
/quadrado {
   moveto
   0 50 rlineto
   50 0 rlineto
   0 -50 rlineto
   closepath
} bind def
```

Note que **quadrado** toma dois valores da pilha, que são utilizados por **moveto**. Uma vez definido, já se pode utilizar o novo operador:




```
0 0 quadrado fill
gsave
   25 25 quadrado .5 setgray fill
   50 50 quadrado
   gsave
      .8 setgray fill
   grestore
   3 setlinewidth stroke
   75 75 quadrado fill
grestore
100 100 quadrado fill
```

Manipulando a pilha de operandos

Existem vários operadores para manipulação da pilha de operandos. Os principais deles são:

`v1` **pop** `-`
Descarta o elemento do topo da pilha:
3 2 1 pop `=> 3 2`

`v1` **dup** `v1 v1`
Duplica o elemento do topo da pilha:
3 2 1 dup `=> 3 2 1 1`

`v1 v2` **exch** `v2 v1`
Permuta os dois elementos do topo da pilha:
2 1 exch `=> 1 2`

`v[n-1] ... v[0] n` **copy** `v[n-1] ... v[0] v[n-1] ... v[0]`
Copia os **n** elementos do topo da pilha (exceto o próprio **n**):
1 2 3 4 3 copy `=> 1 2 3 4 2 3 4`

`v[n-1] ... v[0] n j` **roll** `v[(j-1) mod n] ... v[0] v[n-1] ... v[j mod n]`
Desloca os **n** elementos do topo da pilha **j** posições (no sentido para o topo da pilha).
Se **j** for negativo, o deslocamento será feito no sentido oposto:

```
(a) (b) (c)  3 1  roll  =>  (c) (a) (b)
(a) (b) (c)  3 -1  roll  =>  (b) (c) (a)
(a) (b) (c)  3 2  roll  =>  (b) (c) (a)
```

Operadores aritméticos

Eis alguns operadores aritméticos em *PostScript*:

```
v1 v2  add  (v1+v2)
v1 v2  sub  (v1-v2)
v1 v2  mul  (v1*v2)
v1 v2  div  (v1/v2)
v1 v2  exp  (v1^v2)
v1  sqrt  (v1^(1/2))
v1  abs  |v1|
v1  neg  -v1
```

Operadores lógicos

Seguem–se operadores relacionais e booleanos em *PostScript* (eles retornam um dos valores **true** ou **false**):

```
v1 v2  eq  (v1==v2)
v1 v2  ne  (v1<>v2)
v1 v2  ge  (v1>=v2)
v1 v2  gt  (v1>v2)
v1 v2  le  (v1<=v2)
v1 v2  lt  (v1<v2)
-  true  true  (constante ‘verdadeiro’)
-  false  false  (constante ‘falso’)
bool1 bool2  and  (bool1 AND bool2)
bool1 bool2  or  (bool1 OR bool2)  (‘ou’ inclusivo)
bool1 bool2  xor  (bool1 XOR bool2)  (‘ou’ exclusivo)
bool  not  NOT(bool)
```

Controle de fluxo

Alguns operadores de controle de fluxo importantes em *PostScript* são: **if**, **ifelse**, **repeat** e **for**. A sintaxe é mostrada abaixo:

```
bool proc  if  -
bool proc1 proc2  ifelse  -
n proc  repeat  -
inicial incr limite proc  for  -
```

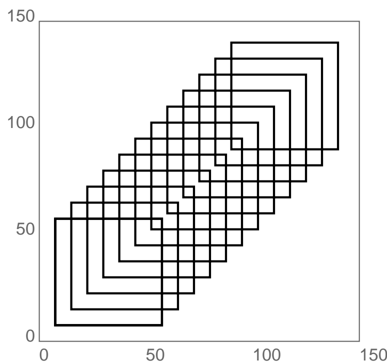
Neste exemplo de **if**, o procedimento entre chaves será executado somente se **x** for maior que 100:

```
x 100 gt {0 0 quadrado stroke} if
```

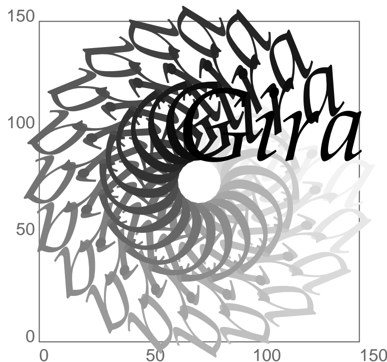
No exemplo de **ifelse**, o procedimento que desenha um quadrado será executado se **x** for maior que 100, caso contrário o procedimento que imprime um texto é que será executado:

```
x 100 gt
   {0 0 quadrado stroke}
   {0 0 moveto /Helvetica findfont 20 scalefont setfont (falso...) show}
ifelse
```

No exemplo a seguir, **repeat** é usado para gerar uma 'cascata' de quadrados. O operador **currentpoint** retorna as coordenadas x e y do ponto atual. Note que **stroke** só necessita ser executado uma vez.




```
gsave
   7.5 7.5 moveto
   13 {
      currentpoint quadrado
      7.5 7.5 translate 0 0 moveto
   } repeat
   stroke
grestore
```



Neste exemplo é ilustrado o uso do laço **for**. A cada iteração, o procedimento recebe o valor da variável do laço, que é usado (depois de ser dividido por 360) na definição do 'tom de cinza'. Essa variável é inicialmente 360, sendo decrementada de 18 a cada interação, até atingir o valor 0.

```
/Palatino-Italic findfont 48 scalefont setfont
gsave
   75 75 translate
   360 -18 0 {
      360 div setgray
      -10 10 moveto (Gira) show
      -18 rotate
   } for
grestore
```

Arrays

Em *PostScript*, *arrays* são delimitados por colchetes, e podem conter quaisquer tipos de dados. Por exemplo, o *array* abaixo contém 6 elementos (uma *string*, dois números, um *array*, um nome literal e um procedimento):

```
[ (texto) 3.14 5 [ (string) 3 ] /literal {quadrado fill} ]
```

Alguns operadores que manipulam *arrays*:

`n` **`array`** `array`
Cria um *array* com n elementos.

`array` **`length`** `n`
Número de elementos do *array*.

`array índice` **`get`** `objeto`
Obtém objeto indexado por índice.

`array índice objeto` **`put`** `–`
Coloca objeto na posição índice do *array*.

`array índice n` **`getinterval`** `subarray`
Obtém subarray com n elementos a partir do índice.

`array1 índice array2` **`putinterval`** `–`
Substitui subarray de array1 (iniciando no índice) pelo array2.

`array proc` **`forall`** `–`
Executa o procedimento proc com cada elemento do array.

À exceção de **array**, todos os operadores acima podem ser também aplicados a *strings*.

Copyright (C) 2004 Hudson Lacerda
This document is released under the terms of the GNU General Public License, Version 2.
Este documento é distribuído segundo os termos da Licença Pública Geral do GNU, Versão 2.