

# PSD

A debugger, interpreter, and viewer for a  
subset of the PostScript language.

J. Scott Drader  
sdrader@shaw.ca

User Manual [v0.710]

<b>1</b>	<b><i>Overview</i></b> .....	<b>3</b>
<b>2</b>	<b><i>Getting started</i></b> .....	<b>4</b>
<b>2.1</b>	<b>Installation</b> .....	<b>4</b>
<b>2.2</b>	<b>Launching PSD</b> .....	<b>4</b>
<b>2.3</b>	<b>Command line options</b> .....	<b>5</b>
<b>3</b>	<b><i>The debug window</i></b> .....	<b>7</b>
<b>3.1</b>	<b>Control/status panel</b> .....	<b>8</b>
<b>3.2</b>	<b>Execution stack panel</b> .....	<b>9</b>
3.2.1	An execution example .....	9
3.2.2	About execution contexts .....	11
3.2.3	Running a context to completion .....	11
3.2.4	Tail recursion.....	12
<b>3.3</b>	<b>Operand stack panel</b> .....	<b>13</b>
<b>3.4</b>	<b>Dictionary watch panel</b> .....	<b>14</b>
<b>3.5</b>	<b>Console</b> .....	<b>14</b>
<b>4</b>	<b><i>Language extensions</i></b> .....	<b>15</b>
<b>4.1</b>	<b>Breakpoints</b> .....	<b>15</b>
<b>4.2</b>	<b>Dictionary watches</b> .....	<b>16</b>
<b>4.3</b>	<b>Output image size/resolution</b> .....	<b>16</b>
<b>5</b>	<b><i>Supported PostScript operators</i></b> .....	<b>17</b>

# 1 Overview

PSD is an interpreter, viewer, and debugger for a subset of the PostScript language. The goal of PSD was not provide a replacement for GhostScript/GSView as a PostScript viewer, but rather to provide a tool to aid early stages of program development.

The interpreter supports a powerful subset of PostScript commands, but certainly does not cover the entire Adobe specification (see section 5 for a list of supported commands). In particular, PSD focuses on the device independent commands, and assumes that the device being rendered to is a computer screen. While PSD does not support the entire PostScript specification, anything that runs in PSD should run in GhostScript/GSView.

PSD is *not* an IDE and does not provide a source code editor. It is thus paired with a good text editor that displays line numbers.

## 2 Getting started

### 2.1 Installation

1. Make sure the Java Runtime Environment is installed. JRE can be obtained at <http://java.sun.com/webapps/getjava/BrowserRedirect?locale=en&host=www.java.com:80>
2. Download the PSD zip file and extract it.
3. (Optional) Create a shortcut to the extracted jar file in a convenient location.

### 2.2 Launching PSD

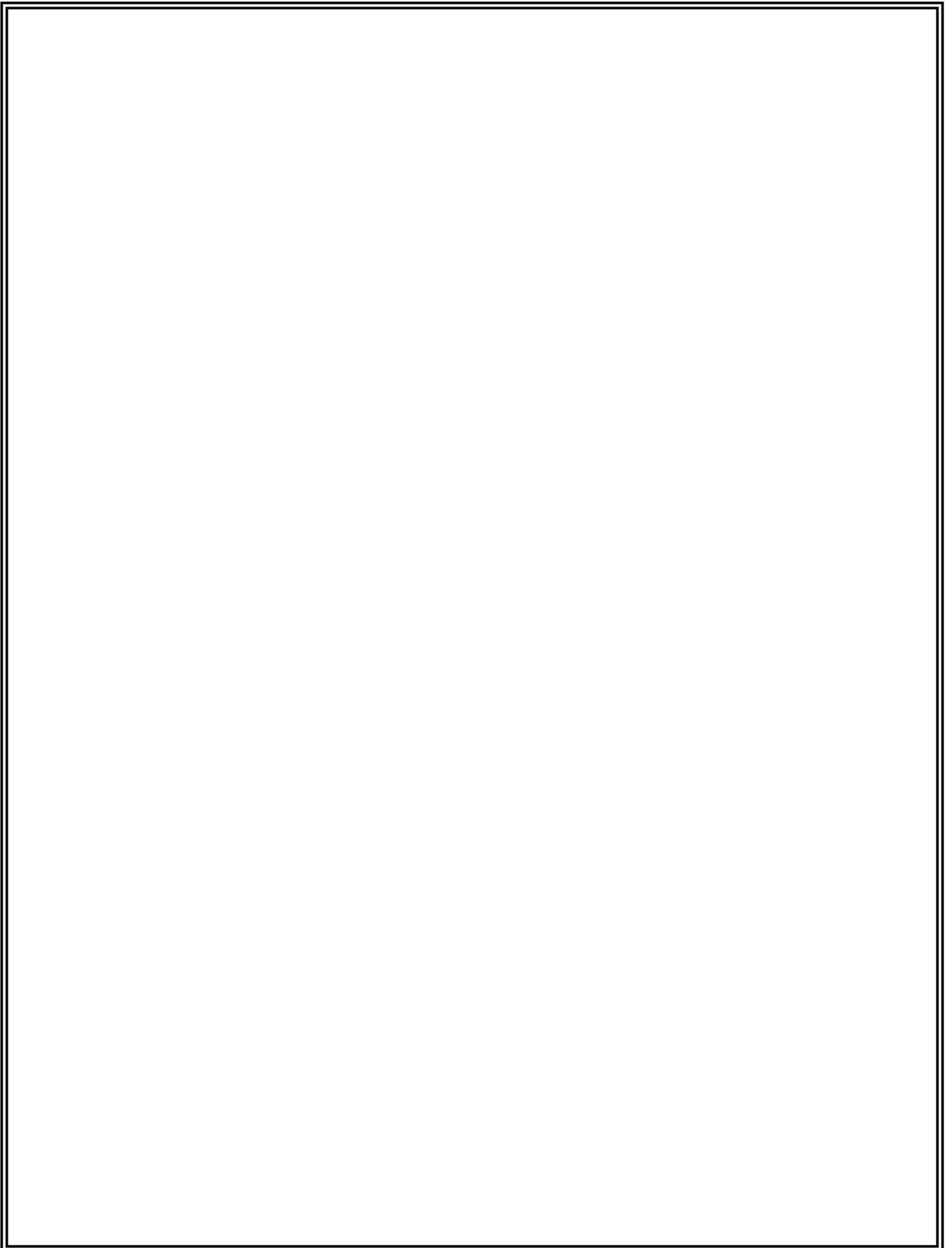
PSD can be launched by simply executing the **psd.jar** file.

By default, PSD displays two windows upon launch; the debug window and the graphics output window. Graphical output defaults to a typical 8.5" by 11" inch (612 by 792 point) page at a resolution of 1.0 pixels per point. These defaults can be overridden by command line options (see the next section).

## 2.3 Command line options

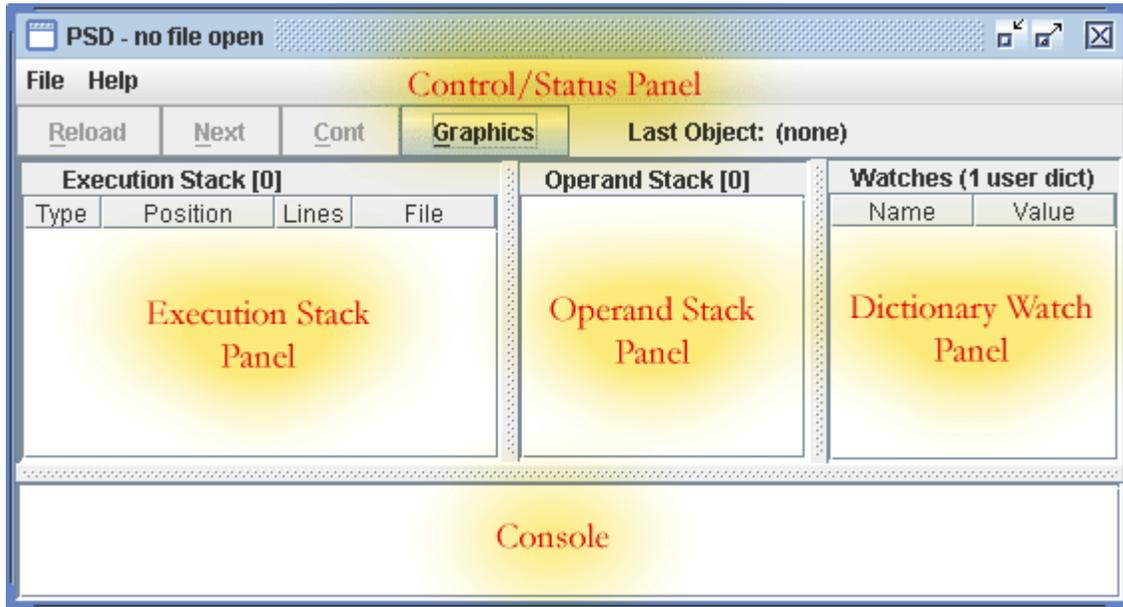
Option	Effect
-fontmap	Specifies location of “Fontmap.GS” file.
-fontdir	Specifies location of <b>.pfb</b> font files.
-hidegfx	PSD launches with graphics window hidden.
-pagesize <width> <height>	Specifies the size of the page in points. Default values are 612 and 792 for width and height respectively.
-res <ratio>	The resolution of the displayed image in pixels per point. The default value is 1.0.

Note that the settings the command line options manipulate can also be changed after start-up. The graphics window can be hidden, and there are language extensions to allow the page size and image resolution to be specified by a program (see section 5).



### 3 The debug window

The debugger window consists of a menu, as well as several components. The components are shown and labeled below:



In the following section, each of the above components is described in terms of what is displayed as well as any interactions that can be performed.

### 3.1 Control/status panel

The menu bar provides the following options:

Menu Option	Effect
File → Open	Displays a file choosing dialog to load a new file.
File → Quit	Exits PSD.

PSD's buttons provide the following functionality:

Button/Command	Effect
Reload	Resets the interpreter and reloads the current file.
Next	Attempts to execute only the next instruction.
Cont	Attempts to execute code until any of the following happen: <ol style="list-style-type: none"><li>1. a new page is reached</li><li>2. a breakpoint is reached</li><li>3. the end of the file is reached</li><li>4. an error occurs</li></ol>
Graphics	Toggles the visibility of the graphics window.

The other element of the control/status panel is the last object, which shows the object most recently executed. An icon representing the object's type is shown followed by a text representation of its value. Following is a table of the icons and their corresponding types.

Icon	Type
	Array
	Boolean
	Dictionary
	Font
	Integer
	Literal Name
	Name
	Null
	Procedure
	Real
	String
	Type

In combination with the execution stack panel, the last object label is useful as an aid to figure out where the interpreter has halted or where an error has occurred.

## 3.2 Execution stack panel

The execution stack panel provides a visualization of the interpreter's execution stack. At any point during execution, it provides a picture of where the interpreter is at in the program, and how it got there.

To fully understand the behavior of the execution stack, it is necessary to have at least a modest understanding of how PostScript execution works. Chapter 3 of the PostScript language reference is a good place to learn more about how execution works. Tracing the execution of simple programs is another effective means of gaining familiarity.

### 3.2.1 An execution example

The example on the following page shows a program illustrating the behavior of the execution stack. The sequence of events is as follows:

1. procedures **proc1** and **proc2** are parsed
2. **proc1** is called
3. **proc1** enters a for loop, iterating the values 0 to 4
4. each iteration passes the loop value to **proc2**
5. **proc2** executes a breakpoint when passed the value 3

The visualization thus shows the state of the execution stack at the breakpoint. The information displayed by the visualization is explained further in the sections following the example.

```

0  % proc1 passes the values the integers 0 through 4 to proc2
1  % (it's a a procedure with 13 objects in it)
2  /proc1 {
3    % this is a loop with a body procedure of 5 objects
4    0 1 5 {
5      /loopVal exch def
6      loopVal proc2
7    } for
8    % the for loop isn't the last object in the procedure
9    (proc1!) =
10 } def
11
12 % proc2 executes a break if its argument is the number 3
13 % this is a procedure with 12 objects
14 /proc2 {
15   /val exch def
16   % the if statement executes a procedure with 1 element
17   val 3 eq {break} if
18   (proc2!) =
19 } def
20
21 proc1

```

Execution Stack [4]			
Type	Position	Lines	File
procedure	item 10 of 12	14-19	exec.ps
for loop	iteration 4, loop value = 3	4-7	exec.ps
procedure	item 11 of 13	2-10	exec.ps
file	line 22		exec.ps

### 3.2.2 About execution contexts

Each row of the table represents a single *execution context*, where the top of the table is the top of the stack. An execution context is a stream of objects for the interpreter to execute. Files, loops, and procedures are examples of contexts.

The information provided about each context includes what type of context it is (Type), how much of it has been executed so far (Position), and the file and location within that file from which it was created (Lines, File).

In the case of an execution context with a fixed number of objects, the Position field shows the number of objects that have been executed so far. In the case of a nested composite objects (a procedure within a procedure for example), the size of the inner context contributes its size to the size of the parent. This is illustrated in the example by the size of **proc1**. Rather than showing length 7 (6 normal objects plus a single composite object), the length is 13 (6 objects plus a composite object of length 7).

The Line field gives the line numbers of the beginning and end of the block of code corresponding to the context. That is, the shown numbers are that of the ‘{’ and ‘}’ delimiters enclosing the body of code that the context executes. Hence in the situation shown below, the **for** loop has its 3 integral arguments on the 20th line, but the ‘{’ delimiter indicating the beginning of its body is on line 21 (and 21 will be the beginning line number shown in the Line field).

20	0 1 10
21	{
22	% some other stuff here
23	} for

### 3.2.3 Running a context to completion

The execution stack panel provides an additional means of executing the loaded program. Double-clicking on an execution context (a row in the table) causes the interpreter to attempt to run until that context is exhausted. This is equivalent to the functionality of the **Cont** command with the completion of the chosen context as additional halting condition. Note that this type of execution is still subject to the regular halting conditions (i.e. the program will still halt before the context is completed if a breakpoint is reached or a **showpage** command is executed).

### 3.2.4 Tail recursion

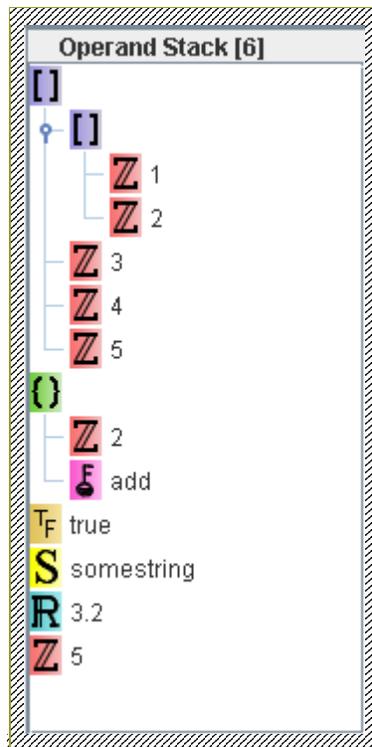
One important concept to understand is that an execution context is removed from the execution stack *before its last object is executed*. This means that if the last action of a procedure is a call to another procedure, the first procedure will not be on the execution stack when the 2nd is executed. This allows tail-recursive processes to run without bloating of the execution stack.

Consider the example program; if **proc1** did not have the display statement at the end, it would not be on the execution stack at the breakpoint (only the loop it contains would be). Furthermore, notice that the small procedure containing only the **break** object is not on the stack at the time the **break** is executed. Instead the top context is a procedure (**proc2**) at the 10<sup>th</sup> of its 12 objects, corresponding to the **if** statement responsible for running the small procedure containing the **break**.

### 3.3 Operand stack panel

The operand stack panel provides a straightforward visualization of the operand stack. Objects on the stack are represented as an icon and text, representing their type and value respectively (see the icon/type table in section 4.1). Arrays/procedures can be expanded by double-clicking the icon. Below is a section of code and an example of the operand stack visualization you can expect to see.

```
5                % an integer
3.2             % a float value
(somestring)    % a string
10 10 eq        % produces a Boolean
{ 2 add }       % a procedure
[ [1 2] 3 4 5 ] % nested arrays
```



### 3.4 Dictionary watch panel

The dictionary watch panel allows tracking of values associated with dictionary keys. The table contains a key/name column (Name), as well as a column that displays the value associated with the key (Value). Values are displayed in a way very similar to that of the operand stack visualization. As with the operand stack, arrays and procedures can be expanded by clicking on their corresponding icons. Watches are added by using the “watch” language extension (described in section 5).

```
/square watch
/pi watch

/square { dup mul } def      % a procedure
/pi 3.14 def                 % a real value

/someName watch             % leave this name undefined
```

Watches (1 user dict)	
Name	Value
/square	 { dup mul }
	—  dup
	—  mul
/pi	 3.14
/someName	undefined

### 3.5 Console

The console is the place where PostScript output is piped to, and also where important messages from the interpreter can be seen. Messages are color coded. The colors correspond to messages as follows:

Color	Message Type
Black	PostScript output (i.e. output from the “pstack” command)
Green	alerts and warnings
Red	errors
Blue	general messages

## 4 Language extensions

PSD extends the PostScript language to allow for standard debugging constructs. The extensions allow for breakpoints, watches, and configuration of the graphical output.

To allow a program to run in both PSD and GhostScript/GSView, the file `psd.inc` can be included at the beginning of the program. It provides empty stubs so the extension commands have no effect on the program when run in GhostScript. The PSD include file is ignored by the PSD run operator, such that the stubs do not overwrite the PSD implementation of commands. Note that using the names of the language extension commands in your program is not recommended.

### 4.1 Breakpoints

The **break** language extension allows one to specify an arbitrary location for the interpreter to halt.

<b>Command</b>	<code>break</code>
<b>Parameters</b>	-
<b>Effect</b>	Halts the interpreter.

A second extension **nbreak** allows the user to “name” the breakpoint so that multiple breakpoints in a program can be easily distinguished.

<b>Command</b>	<code>&lt;name&gt; nbreak</code>
<b>Parameters</b>	name of the breakpoint (literal name)
<b>Effect</b>	Halts the interpreter and displays the name in the console.

Breakpoints are useful in several situations. Placing several breakpoints at intervals in an errant section of code allows one to watch how the interpreter state is affected by each interval of code, and hopefully narrow the potentially problematic section to a smaller section of code.

## 4.2 Dictionary watches

The **watch** command takes a literal name, and adds it to the dictionary watch window. The value associated with the key is updated every time the interpreter stops (i.e. breakpoints, **showpage** commands, errors, etc).

<b>Command</b>	<code>&lt;name&gt; watch</code>
<b>Parameters</b>	name to add (literal name)
<b>Effect</b>	Adds the name to the dictionary watch panel.

## 4.3 Output image size/resolution

Page size and the resolution of the image upon which the page is rendered to on the computer screen can be adjusted by the **pagesize** and **resolution** commands. Both commands reset the interpreter and reload the program if their requested state requires the graphics window to be reinitialized (and are most naturally placed at the beginning of a program).

<b>Command</b>	<code>&lt;width&gt; &lt;height&gt; pagesize</code>
<b>Parameters</b>	desired image width and height in pixels (real or int)
<b>Effect</b>	If the page is currently set to a different size, it is resized, the interpreter is reset, and the current file is reloaded. The command has no effect if the page is already set to the specified size.

<b>Command</b>	<code>&lt;ratio&gt; resolution</code>
<b>Parameters</b>	ratio of pixels to inches, where the default is 1.0 (real or int)
<b>Effect</b>	If the current image resolution is different than requested, the image is reinitialized, the interpreter is reset, and the current file is reloaded. The command has no effect if the image is already set to the specified resolution.

## 5 Supported PostScript operators

[	for	roll
]	forall	rotate
<<	ge	round
=	get	run
==	getinterval	scale
>>	grestore	scafont
abs	gsave	setdash
add	gt	setfont
aload	identmatrix	setgray
and	idiv	setlinecap
arc	if	setlinejoin
arcn	ifelse	setlinewidth
array	index	setmatrix
astore	invertmatrix	setrgbcolor
atan	itransform	shfill
begin	le	show
bind	length	showpage
ceiling	lineto	sin
closepath	ln	sqrt
concat	load	stack
copy	log	stringwidth
cos	loop	stroke
currentlinewidth	lt	sub
currentmatrix	mod	transform
currentpoint	moveto	translate
curveto	mul	truncate
cvi	ne	
cvs	neg	
def	newpath	
dict	not	
div	or	
dup	pop	
end	print	
eq	pstack	
exch	put	
exec	quit	
exit	rand	
exp	rcurveto	
fill	repeat	
findfont	rlineto	
floor	rmoveto	