2. Stack-based Programming

Oscar Nierstrasz

/Times-Roman findfont
18 scalefont
setfont
100 500 moveto
(Hello world) show
showpage

% look up Times Roman font
% scale it to 18 points
% set this to be the current font
% go to coordinate (100, 500)
% draw the string “Hello world”
% render the current page

Hello world
Roadmap

> PostScript objects, types and stacks
> Arithmetic operators
> Graphics operators
> Procedures and and variables
> Arrays and dictionaries
References


> Display Postscript

> GSview for Windows & Linux
Roadmap

- PostScript objects, types and stacks
- Arithmetic operators
- Graphics operators
- Procedures and variables
- Arrays and dictionaries
What is PostScript?

*PostScript “is a simple interpretive programming language ... to describe the appearance of text, graphical shapes, and sampled images on printed or displayed pages.”*

- introduced in 1985 by Adobe
- display standard supported by all major printer vendors
- simple, stack-based programming language
- minimal syntax
- large set of built-in operators
- PostScript programs are usually generated from applications, rather than hand-coded
Postscript variants

> **Level 1:**
  — the original 1985 PostScript

> **Level 2:**
  — additional support for dictionaries, memory management ...

> **Display PostScript:**
  — special support for screen display

> **Level 3:**
  — adds “workflow” support
## Syntax

<table>
<thead>
<tr>
<th><strong>Comments:</strong></th>
<th>from “%” to next newline or formfeed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>% This is a comment</td>
</tr>
<tr>
<td><strong>Numbers:</strong></td>
<td>signed integers, reals and radix numbers</td>
</tr>
<tr>
<td></td>
<td>123 -98 0 +17 -.002 34.5</td>
</tr>
<tr>
<td></td>
<td>123.6e10 1E-5 8#1777 16#FFE 2#1000</td>
</tr>
<tr>
<td><strong>Strings:</strong></td>
<td>text in parentheses or hexadecimal in angle brackets. Special characters are</td>
</tr>
<tr>
<td></td>
<td>escaped: \n \t ( ) ( ) ( ) ( ) ...</td>
</tr>
<tr>
<td><strong>Names:</strong></td>
<td>tokens consisting of “regular characters” but which aren't numbers</td>
</tr>
<tr>
<td></td>
<td>abc Offset $$ 23A 13-456 a.b</td>
</tr>
<tr>
<td></td>
<td>$MyDict @pattern</td>
</tr>
<tr>
<td><strong>Literal names:</strong></td>
<td>start with slash</td>
</tr>
<tr>
<td></td>
<td>/buffer /proc</td>
</tr>
<tr>
<td><strong>Arrays:</strong></td>
<td>enclosed in square brackets</td>
</tr>
<tr>
<td></td>
<td>[ 123 /abc (hello) ]</td>
</tr>
<tr>
<td><strong>Procedures:</strong></td>
<td>enclosed in curly brackets</td>
</tr>
<tr>
<td></td>
<td>{ add 2 div }</td>
</tr>
<tr>
<td></td>
<td>% add top two stack items and divide by 2</td>
</tr>
</tbody>
</table>
Semantics

A PostScript program is a sequence of tokens, representing typed objects, that is interpreted to manipulate the display and **four stacks** that represent the execution state of a PostScript program:

<table>
<thead>
<tr>
<th>Stack Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operand stack:</strong></td>
<td>holds (arbitrary) <strong>operands and results</strong> of PostScript operators</td>
</tr>
<tr>
<td><strong>Dictionary stack:</strong></td>
<td>holds only <strong>dictionaries</strong> where keys and values may be stored</td>
</tr>
<tr>
<td><strong>Execution stack:</strong></td>
<td>holds <strong>executable objects</strong> (e.g. procedures) in stages of execution</td>
</tr>
<tr>
<td><strong>Graphics state stack:</strong></td>
<td>keeps track of <strong>current coordinates</strong> etc.</td>
</tr>
</tbody>
</table>
The first of these stacks is the most important as it is used for all computation.

The dictionary stack is used to encapsulate sets of local variables to be used by procedures we define. The execution stack is mostly hidden from us, and is used by Postscript to manage running procedures. The graphics state stack will make it easy for us to work in different coordinate systems.
Every object is either literal or executable:

Literal objects are pushed on the operand stack:
> integers, reals, string constants, literal names, arrays, procedures

Executable objects are interpreted:
> built-in operators
> names bound to procedures (in the current dictionary context)

Simple Object Types are copied by value
> boolean, fontID, integer, name, null, operator, real ...

Composite Object Types are copied by reference
> array, dictionary, string ...
Roadmap

- PostScript objects, types and stacks
- **Arithmetic operators**
- Graphics operators
- Procedures and variables
- Arrays and dictionaries
The operand stack

Compute the average of 40 and 60: \[40 \ 60 \ \text{add} \ 2 \ \text{div}\]

At the end, the result is left on the top of the operand stack.
Note that numbers are literal objects, so they are pushed on the operand stack, while the operators are executable, so they actually modify the stack.

Aside: note that computation is expressed in RPN — “Reverse Polish Notation” — this is easy to implement without the need for a parser, and was used extensively on HP calculators for this reason.
# Stack and arithmetic operators

<table>
<thead>
<tr>
<th>Stack</th>
<th>Op</th>
<th>New Stack</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>num1 num2</td>
<td>add</td>
<td>sum</td>
<td>num1 + num2</td>
</tr>
<tr>
<td>num1 num2</td>
<td>sub</td>
<td>difference</td>
<td>num1 - num2</td>
</tr>
<tr>
<td>num1 num2</td>
<td>mul</td>
<td>product</td>
<td>num1 * num2</td>
</tr>
<tr>
<td>num1 num2</td>
<td>div</td>
<td>quotient</td>
<td>num1 / num2</td>
</tr>
<tr>
<td>int1 int2</td>
<td>idiv</td>
<td>quotient</td>
<td>integer divide</td>
</tr>
<tr>
<td>int1 int2</td>
<td>mod</td>
<td>remainder</td>
<td>int1 mod int2</td>
</tr>
<tr>
<td>num den</td>
<td>atan</td>
<td>angle</td>
<td>arctangent of num/den</td>
</tr>
<tr>
<td>any</td>
<td>pop</td>
<td>-</td>
<td>discard top element</td>
</tr>
<tr>
<td>any1 any2</td>
<td>exch</td>
<td>any2 any1</td>
<td>exchange top two elements</td>
</tr>
<tr>
<td>any</td>
<td>dup</td>
<td>any any</td>
<td>duplicate top element</td>
</tr>
<tr>
<td>any1 ... anyn n</td>
<td>copy</td>
<td>any1 ... anyn any1 ... anyn</td>
<td>duplicate top n elements</td>
</tr>
<tr>
<td>anyn ... any0 n</td>
<td>index</td>
<td>anyn ... any0 anyn</td>
<td>duplicate n+1th element</td>
</tr>
</tbody>
</table>

*and many others...*
Roadmap

- PostScript objects, types and stacks
- Arithmetic operators
- **Graphics operators**
- Procedures and variables
- Arrays and dictionaries
Coordinates

Coordinates are measured in points:

72 points = 1 inch = 2.54 cm.
Drawing a Box

“A path is a set of straight lines and curves that define a region to be filled or a trajectory that is to be drawn on the current page.”

newpath % clear the current drawing path
100 100 moveto % move to (100,100)
100 200 lineto % draw a line to (100,200)
200 200 lineto
200 100 lineto
100 100 lineto
10 setlinewidth % set width for drawing
stroke % draw along current path
showpage % and display current page
If you have a computer that directly supports Display Postscript, you can execute these examples without sending them to a printer. Alternatively you may use a dedicated open source program, such as Ghostscript.

Why is the bottom left corner not perfectly closed? Simulate what postscript is doing with a pen of 10 points width.
### Path construction operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>newpath</strong></td>
<td>Initialize current path to be empty</td>
</tr>
<tr>
<td><strong>currentpoint</strong></td>
<td>Return current coordinates</td>
</tr>
<tr>
<td>x y <strong>moveto</strong></td>
<td>Set current point to (x, y)</td>
</tr>
<tr>
<td>dx dy <strong>rmoveto</strong></td>
<td>Relative moveto</td>
</tr>
<tr>
<td>x y <strong>lineto</strong></td>
<td>Append straight line to (x, y)</td>
</tr>
<tr>
<td>dx dy <strong>rlineto</strong></td>
<td>Relative lineto</td>
</tr>
<tr>
<td>x y r ang1 ang2 <strong>arc</strong></td>
<td>Append counterclockwise arc</td>
</tr>
<tr>
<td><strong>closepath</strong></td>
<td>Connect subpath back to start</td>
</tr>
<tr>
<td><strong>fill</strong></td>
<td>-</td>
</tr>
<tr>
<td><strong>stroke</strong></td>
<td>Draw line along current path</td>
</tr>
<tr>
<td><strong>showpage</strong></td>
<td>Output and reset current page</td>
</tr>
</tbody>
</table>

**Others:** arcn, arcto, curveto, rcurveto, flattenpath, ...
“Hello World” in Postscript

Before you can print text, you must
1. look up the desired font,
2. scale it to the required size, and
3. set it to be the current font.

```
/Times-Roman findfont
  18 scalefont
  setfont
100 500 moveto
(Hello world) show
showpage

% look up Times Roman font
% scale it to 18 points
% set this to be the current font
% go to coordinate (100, 500)
% draw the string “Hello world”
% render the current page
```

Hello world
Note that /Times-Roman and (Hello world) are literal objects, so are pushed on the stack, not executed.
Encapsulated PostScript

EPSF is a standard format for importing and exporting PostScript files between applications.

```
%!PS-Adobe-3.0 EPSF-3.0
%!BoundingBox: 90 490 200 520
/Times-Roman findfont
  18 scalefont
  setfont
100 500 moveto
(Hello world) show
showpage
```

(90, 490)

Hello world

(200, 520)
## Character and font operators

<table>
<thead>
<tr>
<th>key</th>
<th>findfont</th>
<th>font</th>
<th>return font dict identified by key</th>
</tr>
</thead>
<tbody>
<tr>
<td>font scale</td>
<td>scalefont</td>
<td>font’</td>
<td>scale font by given scale to produce font'</td>
</tr>
<tr>
<td>font</td>
<td>setfont</td>
<td>-</td>
<td>set font dictionary</td>
</tr>
<tr>
<td>-</td>
<td>currentfont</td>
<td>font</td>
<td>return current font</td>
</tr>
<tr>
<td>string</td>
<td>show</td>
<td>-</td>
<td>print string</td>
</tr>
<tr>
<td>string</td>
<td>stringwidth</td>
<td>wx</td>
<td>width of string in current font</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wy</td>
<td></td>
</tr>
</tbody>
</table>

**Others:** definefont, makefont, FontDirectory, StandardEncoding ....
Roadmap

- PostScript objects, types and stacks
- Arithmetic operators
- Graphics operators
- Procedures and variables
- Arrays and dictionaries
Variables and procedures are defined by binding names to literal or executable objects.

Define a general procedure to compute averages:

```
def /average { add 2 div } def
% bind the name “average” to “{ add 2 div }”
40 60 average
```

<table>
<thead>
<tr>
<th>key value</th>
<th>def</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>associate key and value in current dictionary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Note that once the literal /average is defined, average becomes an executable operator.
A Box procedure

Most PostScript programs consist of a *prologue* and a script.

```postscript
% Prologue -- application specific procedures
/box {
  % grey x y -> __
  newpath
  moveto % x y -> __
  0 150 rlineto % relative lineto
  150 0 rlineto
  0 -150 rlineto
  closepath % cleanly close path!
  setgray % grey -> __
  fill % colour in region
} def
% Script -- usually generated
0 100 100 box
0.4 200 200 box
0.6 300 300 box
0 setgray % set drawing color back to black!
showpage
```
Postscript programs are typically generated by document authoring systems. The programs they generate consist of prologues that were originally hand-written, and scripts that are generated.
# Graphics state and coordinate operators

<table>
<thead>
<tr>
<th>num</th>
<th>setlinewidth</th>
<th>-</th>
<th>set line width</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>setgray</td>
<td>-</td>
<td>set colour to gray value</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(0 = black; 1 = white)</td>
</tr>
<tr>
<td>sx sy</td>
<td>Scale</td>
<td>-</td>
<td>scale user space by sx and sy</td>
</tr>
<tr>
<td>angle</td>
<td>rotate</td>
<td>-</td>
<td>rotate user space by angle degrees</td>
</tr>
<tr>
<td>tx ty</td>
<td>translate</td>
<td>-</td>
<td>translate user space by (tx, ty)</td>
</tr>
<tr>
<td></td>
<td>matrix</td>
<td>-</td>
<td>create identity matrix</td>
</tr>
<tr>
<td>matrix</td>
<td>currentmatrix</td>
<td>matrix</td>
<td>fill matrix with CTM</td>
</tr>
<tr>
<td>matrix</td>
<td>setmatrix</td>
<td>matrix</td>
<td>replace CTM by matrix</td>
</tr>
<tr>
<td></td>
<td>gsave</td>
<td>-</td>
<td>save graphics state</td>
</tr>
<tr>
<td></td>
<td>grestore</td>
<td>-</td>
<td>restore graphics state</td>
</tr>
</tbody>
</table>

`gsave` saves the current path, gray value, line width and user coordinate system.
The graphics state operators make it easy to work in a simple coordinate system, even if the target is scaled or rotated: instead of drawing a rotated square, you can draw a regular square in a rotated coordinate system.
A Fibonacci Graph

```
/fibInc { % m n -> n (m+n)
  exch % m n -> n m
  1 index % n m -> n m n
  add % m n -> n (m+n)
} def
/x 0 def /y 0 def /dx 10 def
newpath
100 100 translate % make (100, 100) the origin
x y moveto % i.e., relative to (100, 100)
0 1
25 {
  /x x dx add def % increment x
  dup /y exch 100 idiv def % set y to 1/100 last fib val
  x y lineto % draw segment
  fibInc
} repeat
2 setlinewidth
stroke
showpage
```
Numbers and Strings

Numbers and other objects must be converted to strings before they can be printed:

<table>
<thead>
<tr>
<th>int</th>
<th>string</th>
<th>string</th>
<th>create string of capacity int</th>
</tr>
</thead>
<tbody>
<tr>
<td>any string</td>
<td>cvs</td>
<td>substring</td>
<td>convert to string</td>
</tr>
</tbody>
</table>
Factorial

/LM 100 def % left margin
/FS 18 def % font size
/sBuf 20 string def % string buffer of length 20
/fact { % n -> n!
    dup 1 lt % n -> n bool
    { pop 1 }
    {
        dup % n -> n n
        1
        sub
        fact
        mul
    }
    ifelse
} def
/showInt { % n -> __
    sBuf cvs show % convert an integer to a string and show it
} def
Factorial ...

/showFact { % n -> __
    dup showInt
    (! = ) show
    fact showInt
} def

/newline { % __ -> __
    currentpoint exch pop
    FS 2 add sub
    LM exch moveto
} def

/Times-Roman findfont FS scalefont setfont
LM 600 moveto
0 1 20 { showFact newline } for % do from 0 to 20
showpage

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6.22702e+09
14! = 8.71783e+10
15! = 1.30767e+12
16! = 2.09228e+13
17! = 3.55687e+14
18! = 6.40237e+15
19! = 1.21645e+17
20! = 2.4329e+18
## Boolean, control and string operators

<table>
<thead>
<tr>
<th>any1 any2</th>
<th>eq</th>
<th>bool</th>
<th>test equal</th>
</tr>
</thead>
<tbody>
<tr>
<td>any1 any2</td>
<td>ne</td>
<td>bool</td>
<td>test not equal</td>
</tr>
<tr>
<td>any1 any2</td>
<td>ge</td>
<td>bool</td>
<td>test greater or equal</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
<td>push boolean value true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>push boolean value false</td>
</tr>
<tr>
<td>bool proc</td>
<td>if</td>
<td>-</td>
<td>execute proc if bool is true</td>
</tr>
<tr>
<td>bool proc1 proc2</td>
<td>ifelse</td>
<td>-</td>
<td>execute proc1 if bool is true else proc2</td>
</tr>
<tr>
<td>init incr limit proc</td>
<td>for</td>
<td>-</td>
<td>execute proc with values init to limit by steps of incr</td>
</tr>
<tr>
<td>int proc</td>
<td>repeat</td>
<td>-</td>
<td>execute proc int times</td>
</tr>
<tr>
<td>string</td>
<td>length</td>
<td>int</td>
<td>number of elements in string</td>
</tr>
<tr>
<td>string index</td>
<td>get</td>
<td>int</td>
<td>get element at position index</td>
</tr>
<tr>
<td>string index int</td>
<td>put</td>
<td>-</td>
<td>put int into string at position index</td>
</tr>
<tr>
<td>string proc</td>
<td>forall</td>
<td>-</td>
<td>execute proc for each element of string</td>
</tr>
</tbody>
</table>
A simple formatter

/LM 100 def % left margin
/RM 250 def % right margin
/FS 18 def % font size
/showStr { % string -> __
    dup stringwidth pop % get (just) string’s width
    currentpoint pop % current x position
    add % where printing would bring us
    RM gt { newline } if % newline if this would overflow RM
    show
} def
/newline { % __ -> __
    currentpoint exch pop % get current y
    FS 2 add sub % subtract offset
    LM exch moveto % move to new x y
} def
/format { { showStr ( ) show } forall } def % array -> __
/Times-Roman findfont FS scalefont setfont
LM 600 moveto
Now is the time for all good men to come to the aid of the party.
Roadmap

> PostScript objects, types and stacks
> Arithmetic operators
> Graphics operators
> Procedures and variables
> Arrays and dictionaries
### Array and dictionary operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Type</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>mark obj0 ... objn-1</td>
<td>[ ] mark start array construction</td>
<td></td>
<td>start array construction</td>
</tr>
<tr>
<td>int</td>
<td>array int array</td>
<td>array</td>
<td>create array of length n</td>
</tr>
<tr>
<td>array</td>
<td>length int any array</td>
<td>any</td>
<td>get element at index position</td>
</tr>
<tr>
<td>array index</td>
<td>get any array index</td>
<td>any</td>
<td>get element at index position</td>
</tr>
<tr>
<td>array index any</td>
<td>put - put element at index position</td>
<td></td>
<td>put element at index position</td>
</tr>
<tr>
<td>array proc</td>
<td>forall - execute proc for each array element</td>
<td></td>
<td>execute proc for each array element</td>
</tr>
<tr>
<td>int</td>
<td>dict dict create dictionary of capacity int</td>
<td>dict</td>
<td>create dictionary of capacity int</td>
</tr>
<tr>
<td>dict</td>
<td>length int dict length</td>
<td>int</td>
<td>number of key-value pairs</td>
</tr>
<tr>
<td>dict</td>
<td>maxlength int dict maxlength</td>
<td>int</td>
<td>capacity</td>
</tr>
<tr>
<td>dict</td>
<td>begin - push dict on dict stack</td>
<td></td>
<td>push dict on dict stack</td>
</tr>
<tr>
<td>-</td>
<td>end - pop dict stack</td>
<td></td>
<td>pop dict stack</td>
</tr>
</tbody>
</table>
Using Dictionaries — Arrowheads

\[
/\text{arrowdict} 14 \text{ dict def} \quad \% \text{ make a new dictionary}
\]
arrowdict begin
  \[
/m\text{trx} \text{ matrix def} \quad \% \text{ allocate space for a matrix}
\]
etd
/\text{arrow} \{ \\
  arrowdict begin \quad \% \text{ open the dictionary}
  /\text{headlength} \text{ exch def} \quad \% \text{ grab args}
  /\text{halfheadthickness} \text{ exch 2 div def}
  /\text{halfthickness} \text{ exch 2 div def}
  /\text{tippy} \text{ exch def}
  /\text{tipx} \text{ exch def}
  /\text{taily} \text{ exch def}
  /\text{tailx} \text{ exch def}
  /\text{dx} \text{ tipx tailx sub def}
  /\text{dy} \text{ tippy taily sub def}
  /\text{arrowlength} \text{ dx dx mul dy dy mul add sqrt def}
  /\text{angle} \text{ dy dx atan def}
  /\text{base} \text{ arrowlength headlength sub def}
  /\text{savematrix} \text{ mtrx currentmatrix def} \quad \% \text{ save the coordinate system}
\]

Usage: tailx taily tipx tipy \text{ thickness headthickness headlength arrow}
mtrx currentmatrix
pop
tailx taily translate
angle rotate
0 halfthickneg moveto
base halfthickneg lineto
base halfheadthickneg lineto
arrowlength 0 lineto
base halfheadthick lineto
base halfthick lineto
0 halfthick lineto
closepath
savematrix setmatrix
end
} def

% save the coordinate system
% translate to start of arrow
% rotate coordinates
% draw as if starting from (0,0)
% restore coordinate system
Notice how a dictionary is used to allocate space for all the “local” variables of the arrow procedure. We need 14 slots for 14 key-value pairs (7 parameters plus another 7 “local variables”). By defining our own dictionary, and pushing it to the dictionary stack, we make sure that the names we use do not conflict with any other similar names used by other procedures. The dictionary stack therefore serves the same purpose as the runtime stack in most programming languages.
Instantiating Arrows

Usage:

\texttt{\texttt{tailx taily tipx tipy thickness headthickness headlength arrow}}

```
newpath
  318 340 72 340 10 30 72 arrow
fill
newpath
  382 400 542 560 72 232 116 arrow
3 setlinewidth stroke
newpath
  400 300 400 90 90 200 200 3 sqrt mul 2 div arrow
.65 setgray fill
showpage
```
NB: arrow does not do a newpath, so arrows can be added to existing paths
What you should know!

- What kinds of stacks does PostScript manage?
- When does PostScript push values on the operand stack?
- What is a path, and how can it be displayed?
- How do you manipulate the coordinate system?
- Why would you define your own dictionaries?
- How do you compute a bounding box for your PostScript graphic?
Can you answer these questions?

- How would you implement a `while` procedure?
- When should you use `translate` instead of `moveto`?
- How could you use dictionaries to simulate object-oriented programming?
- How would you program this graphic?
Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:
- **Share** — copy and redistribute the material in any medium or format
- **Adapt** — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

**Under the following terms:**

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

- **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

[http://creativecommons.org/licenses/by-sa/4.0/](http://creativecommons.org/licenses/by-sa/4.0/)