b
UNIVERSITÄT
BERN

2. Stack-based Programming

Oscar Nierstrasz

```
/Times-Roman findfont          % look up Times Roman font
    18 scalefont               % scale it to 18 points
    setfont                    % set this to be the current font
100 500 moveto                 % go to coordinate (100, 500)
(Hello world) show             % draw the string "Hello world"
showpage                       % render the current page
```

Hello world

Roadmap

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

References

- *PostScript® Language Tutorial and Cookbook*, Adobe Systems Incorporated, Addison-Wesley, 1985
- *PostScript® Language Reference Manual*, Adobe Systems Incorporated, second edition, Addison-Wesley, 1990
- Display Postscript
  - http://en.wikipedia.org/wiki/Display_PostScript
- GSview for Windows & Linux
  - http://www.ghostscript.com/GSview.html

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

Although Postscript has been around for a while, it has been extremely successful, having established itself as the de facto standard for printers. Although hardly anyone programs in Postscript, programs are generated every time anyone prints a document.

The language is interesting to study as an example of a powerful and expressive stack-based language.

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

Comments:	**from "%" to next newline or formfeed** `% This is a comment`

Numbers:	**signed integers, reals and radix numbers**
	`123 -98 0 +17 -.002 34.5`
	`123.6e10 1E-5 8#1777 16#FFE 2#1000`

Strings:	**text in parentheses or hexadecimal in angle brackets. Special characters are escaped:** `\n \t \( \) \\ …)`

Names:	**tokens consisting of "regular characters" but which aren't numbers**
	`abc Offset $$ 23A 13-456 a.b`
	`$MyDict @pattern`

Literal names:	**start with slash** `/buffer /proc`
Arrays:	**enclosed in square brackets** `[ 123 /abc (hello) ]`

Procedures:	**enclosed in curly brackets**
	`{ add 2 div }`
	`% add top two stack items and divide by 2`

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:

Operand stack:	holds (arbitrary) *operands and results* of PostScript operators
Dictionary stack:	holds only *dictionaries* where keys and values may be stored
Execution stack:	holds *executable objects* (e.g. procedures) in stages of execution
Graphics state stack:	keeps track of *current coordinates* etc.

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.

Object types

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 ...

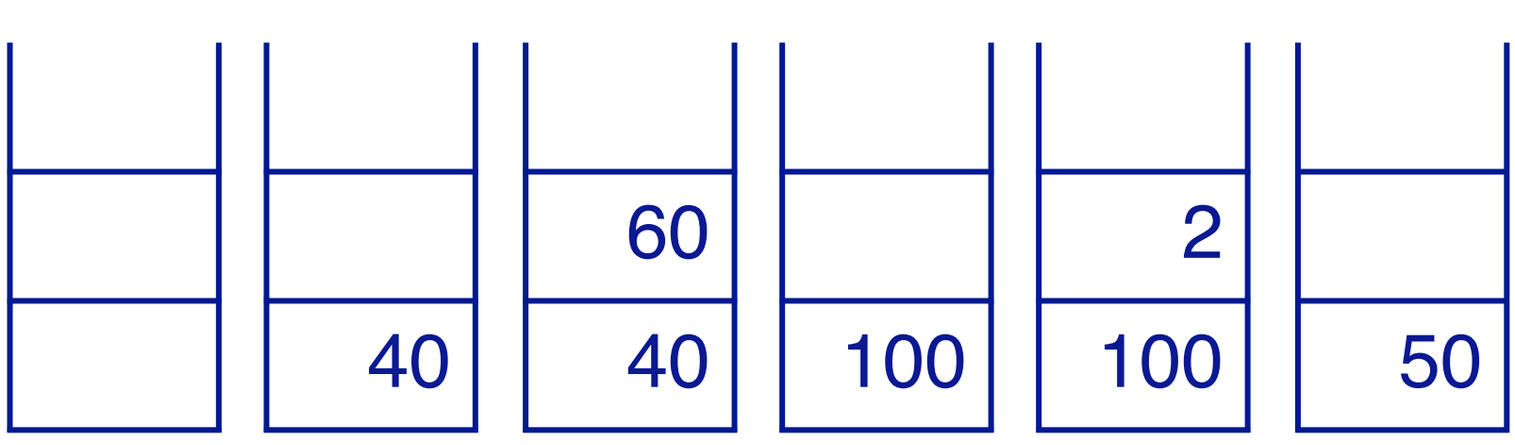
Roadmap

- 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 add 2 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

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

and many others ...

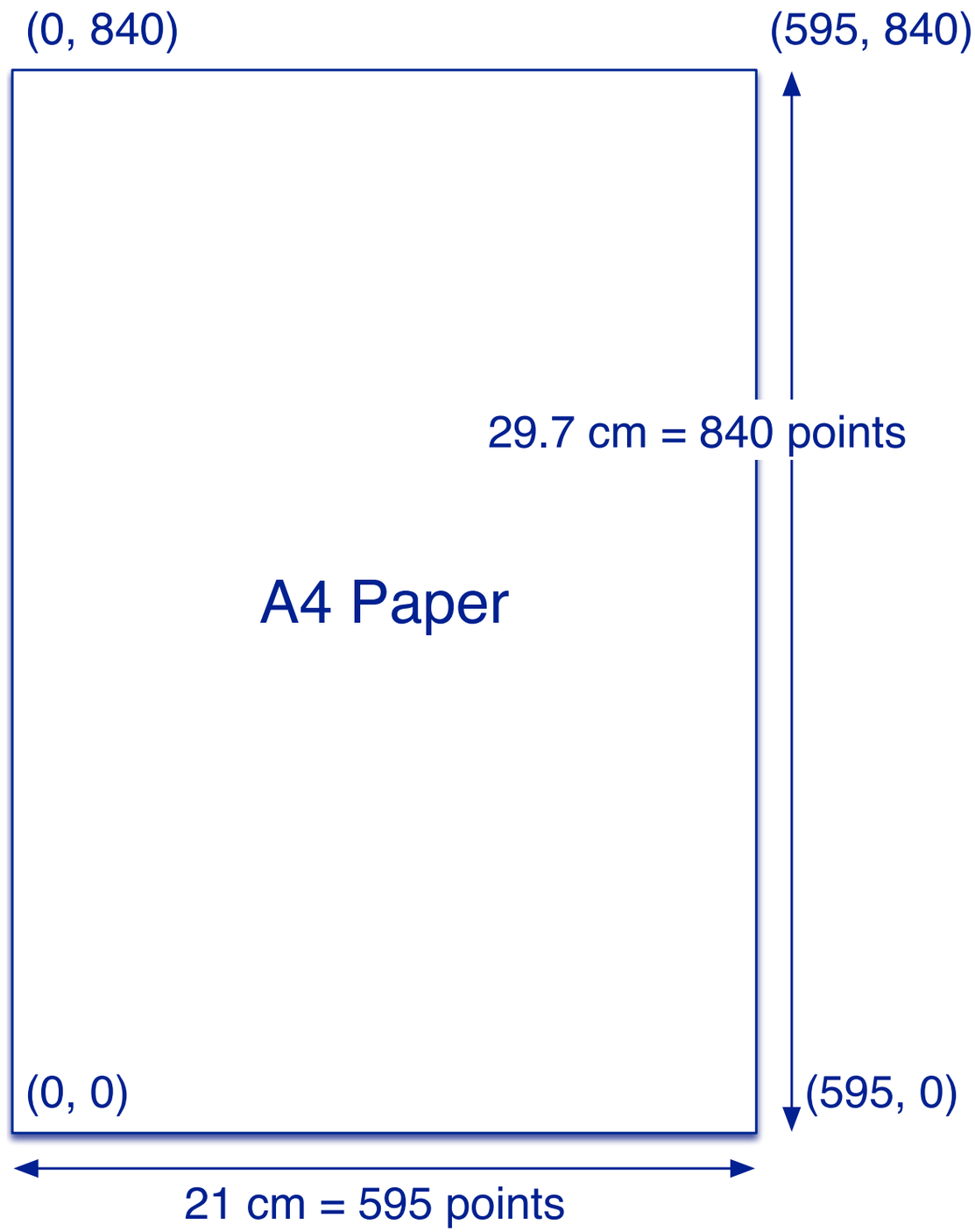
Roadmap

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

Coordinates

Coordinates are measured in points:

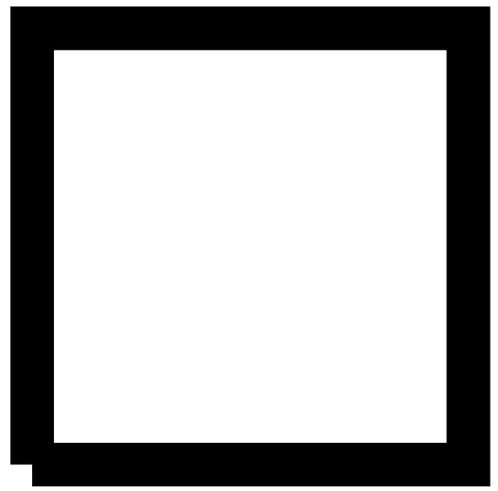
72 points = 1 inch
= 2.54 cm.



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

-	**`newpath`**	-	initialize current path to be empty
-	**`currentpoint`**	x y	return current coordinates
x y	**`moveto`**	-	set current point to (x, y)
dx dy	**`rmoveto`**	-	relative `moveto`
x y	**`lineto`**	-	append straight line to (x, y)
dx dy	**`rlineto`**	-	relative `lineto`
x y r ang1 ang2	**`arc`**	-	append counterclockwise arc
-	**`closepath`**	-	connect subpath back to start
-	**`fill`**	-	
-	**`stroke`**	-	draw line along current path
-	**`showpage`**	-	output and reset current page

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

(200, 520)

Hello world

(90, 490)

Character and font operators

key	**`findfont`**	font	return font dict identified by key
font scale	**`scalefont`**	font’	scale font by given scale to produce font’
font	**`setfont`**	-	set font dictionary
-	**`currentfont`**	font	return current font
string	**`show`**	-	print string
string	**`stringwidth`**	wx wy	width of string in current font

Others: `definefont`, `makefont`, `FontDirectory`, `StandardEncoding` ....

Roadmap

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

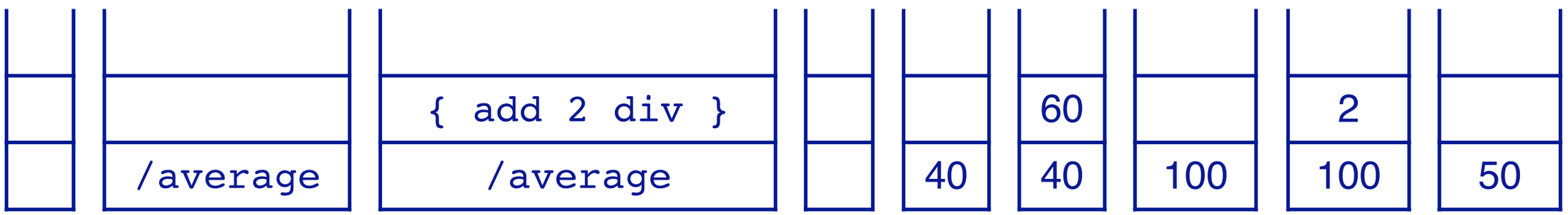
Procedures and Variables

Variables and procedures are defined by binding names to literal or executable objects.

key value	**`def`**	-	associate key and value in current dictionary

Define a general procedure to compute averages:

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

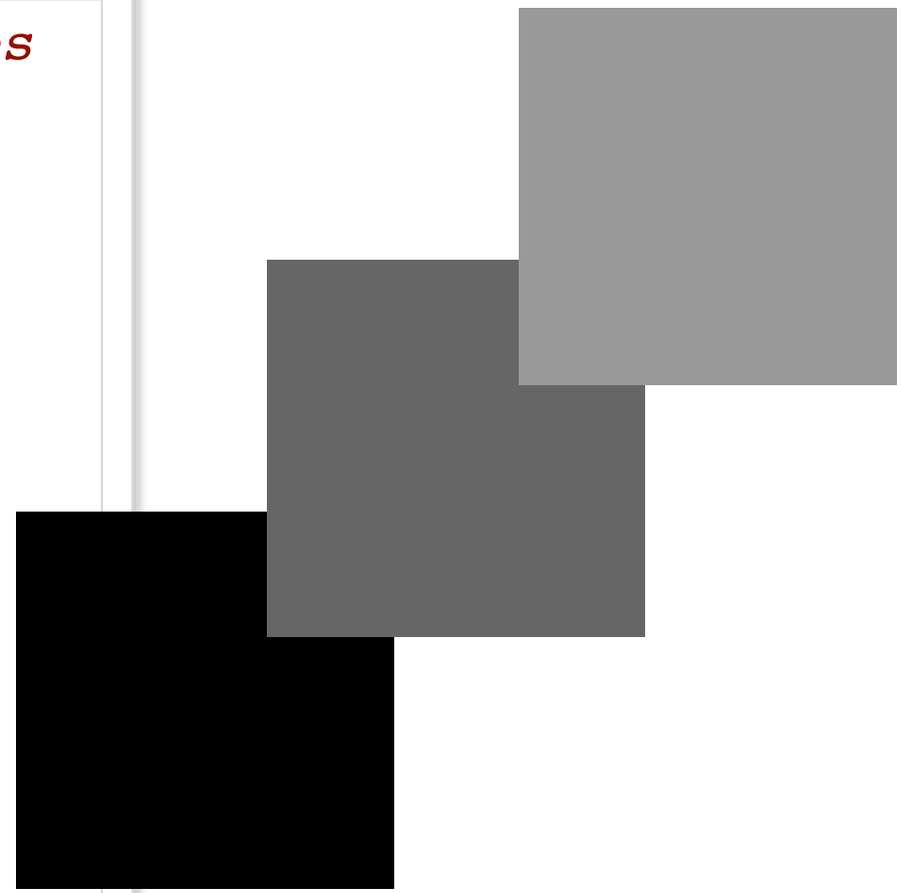


Note that once the literal `/average` is defined, `average` becomes an executable operator.

A Box procedure

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

```
% 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

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

`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:

int	**`string`**	string	create string of capacity int
any string	**`cvs`**	substring	convert to string

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 }                   % 0 -> 1
   {
      dup                      % n -> n n
      1                        % -> n n 1
      sub                      % -> n (n-1)
      fact                     % -> n (n-1)!NB: recursive lookup
      mul                      % n!
   }
   ifelse
} def
/showInt {                     % n -> __
   sBuf cvs show               % convert an integer to a string and show it
} def
```

Factorial ...

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

/newline {                           % __ -> __
   currentpoint exch pop             % get current y
   FS 2 add sub                      % subtract offset
   LM exch moveto                    % move to new x y
} 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

any1 any2	**`eq`**	bool	test equal
any1 any2	**`ne`**	bool	test not equal
any1 any2	**`ge`**	bool	test greater or equal
-	**`true`**	true	push boolean value true
-	**`false`**	false	push boolean value false
bool proc	**`if`**	-	execute proc if bool is true
bool proc1 proc2	**`ifelse`**	-	execute proc1 if bool is true else proc2
init incr limit proc	**`for`**	-	execute proc with values init to limit by steps of incr
int proc	**`repeat`**	-	execute proc int times
string	**`length`**	int	number of elements in string
string index	**`get`**	int	get element at position index
string index int	**`put`**	-	put int into string at position index
string proc	**`forall`**	-	execute proc for each element of string

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
```

A simple formatter ...

```
[ (Now) (is) (the) (time) (for) (all) (good) (men) (to)
(come) (to) (the) (aid) (of) (the) (party.) ] format
showpage
```

Now is the time for
all good men to
come to the aid of
the party.

Roadmap

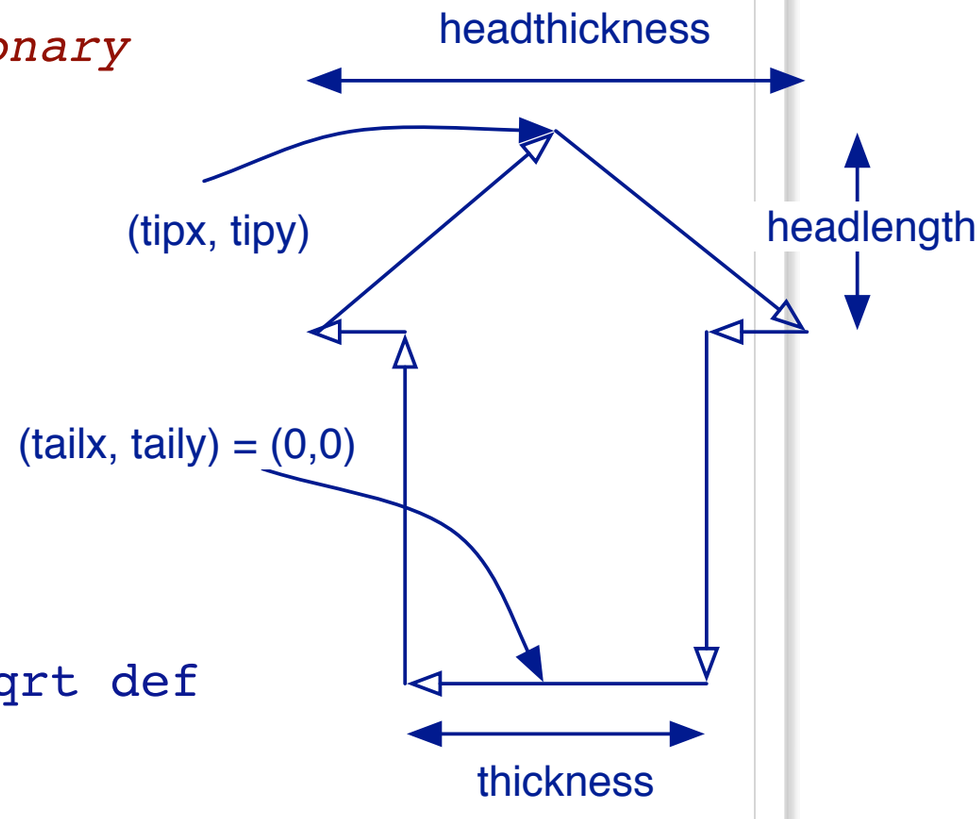
- PostScript objects, types and stacks
- Arithmetic operators
- Graphics operators
- Procedures and variables
- **Arrays and dictionaries**

Array and dictionary operators

-	**[**	mark	start array construction
mark obj0 ... objn-1	**]**	array	end array construction
int	**array**	array	create array of length n
array	**length**	int	number of elements in array
array index	**get**	any	get element at index position
array index any	**put**	-	put element at index position
array proc	**forall**	-	execute proc for each array element
int	**dict**	dict	create dictionary of capacity int
dict	**length**	int	number of key-value pairs
dict	**maxlength**	int	capacity
dict	**begin**	-	push dict on dict stack
-	**end**	-	pop dict stack

Using Dictionaries — Arrowheads

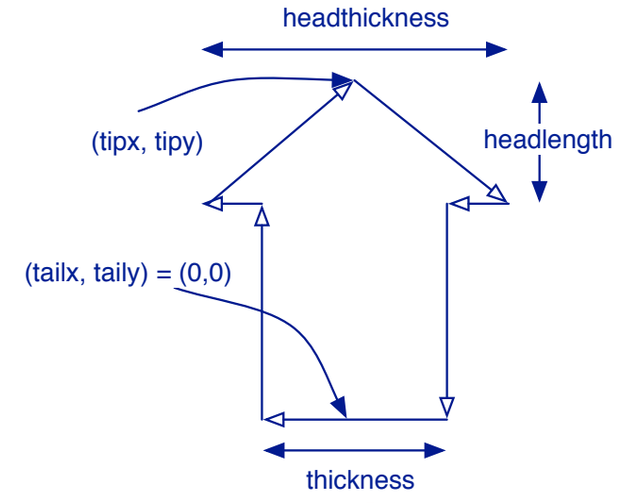
```
/arrowdict 14 dict def          % make a new dictionary
arrowdict begin
   /mtrx matrix def             % allocate space for a matrix
end
/arrow {
   arrowdict begin              % open the dictionary
      /headlength exch def  % grab args
      /halfheadthickness exch 2 div def
      /halfthickness exch 2 div def
      /tipy exch def
      /tipx exch def
      /taily exch def
      /tailx exch def
      /dx tipx tailx sub def
      /dy tipy taily sub def
      /arrowlength dx dx mul dy dy mul add sqrt def
      /angle dy dx atan def
      /base arrowlength headlength sub def
      /savematrix mtrx currentmatrix def % save the coordinate system
```



Usage: tailx taily tipx tipy *thickness headthickness* headlength **arrow**

```
        tailx taily translate                   % translate to start of arrow
        angle rotate                            % rotate coordinates

        0 halfthickness neg moveto              % draw as if starting from (0,0)
        base halfthickness neg lineto
        base halfheadthickness neg lineto
        arrowlength 0 lineto
        base halfheadthickness lineto
        base halfthickness lineto
        0 halfthickness lineto
        closepath
        savematrix setmatrix                    % restore coordinate system
    end
} def
```

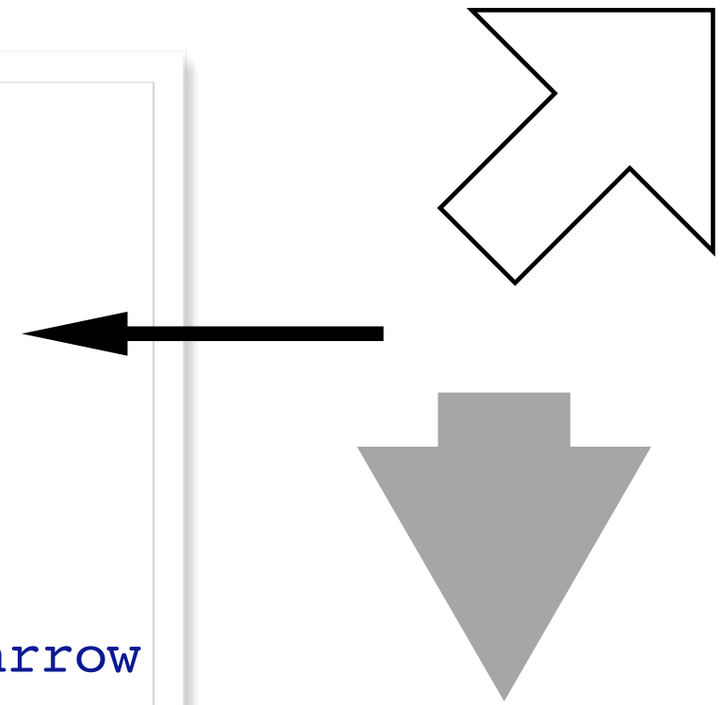


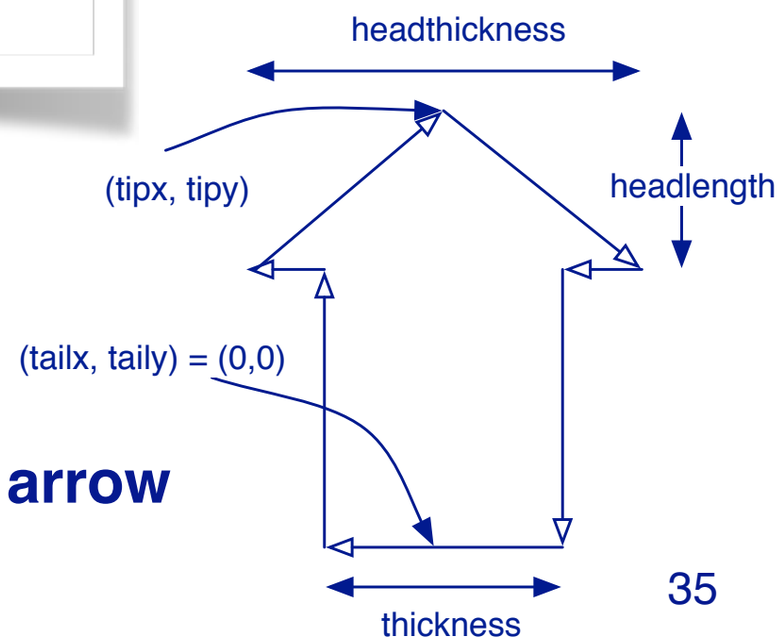
Notice how a dictionary is used to allocate space for all the "local" variables of the arrow procedure. We need 14 slots for 14 key-value pairs (7 parameters plus another 7 "local variables"). By defining our own dictionary, and pushing it to the dictionary stack, we make sure that the names we use do not conflict with any other similar names used by other procedures.

The dictionary stack therefore serves the same purpose as the run-time stack in most programming languages.

Instantiating Arrows

```
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
```





Usage: tailx taily tipx tipy *thickness headthickness* headlength **arrow**

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?*