PicoScript specification

Christopher League*

PostScript® is a language developed by Adobe Systems starting in the early 1980s. It is an important language in printing and publishing, used to describe fonts, graphics, and text. Its imaging model and many of its operators are also used in Adobe's Portable Document Format (PDF). But PostScript is a full Turing-Machine–equivalent programming language, not just a document format. The language is described in detail in The PostScript Language Reference (PDF).

This document describes **PicoScript,** so named because it's a small subset of PostScript. It contains enough of the basic syntax and operators to help us understand how a **stack-based** language works, and to give us experience writing a lexical analyzer and interpreter. If you'd like to experiment with the real language, see the section Running PostScript below.

Overview

I said above that PicoScript is a stack-based language. In this section, we'll explore what that means. Expressions are in **postfix** notation (also called RPN) and they are evaluated using an **operand stack.** In algebra and many programming languages, we use **infix** notation for arithmetic, which means that the operator appears *in between* the two operands, as in `x + 3`. Lots of programming languages also use **prefix** notation for function calls, as in `pow(x,3)` to compute x^3. With postfix, the operator appears *after* the operands. Also, PicoScript uses *words* for arithmetic operators rather than symbols. So here is an integer expression:

```
3 8 2 mul add 4 sub
```

where the `mul` is short for multiply, and `sub` is subtract. The integers are operands and the words are operators. Each operand is pushed onto the **operand stack,** in order. The first three tokens are integers, so after reading each one our operand stack contains:

```
[3, 8, 2]
```

(The top of the stack is to the right.) Now we read the `mul` operator. It pops the **top two** elements off the stack (8 and 2), multiplies them (16), and pushes the result onto the stack:

```
[3, 16]
```

The next operator in our input is `add`. Again, it pops the **top two** elements off the stack (3 and 16), adds them (19), and pushes the result onto the stack:

```
[19]
```

Next our input contained the integer `4`, so we just push that as before:

```
[19, 4]
```

Finally, the last operator `sub` works as before. But now we need to be cautious because subtraction is not commutative: it matters which order we pop. The way it works in PicoScript is that they obey the left-to-right order as shown, meaning the second operand to subtraction is actually the *top* of the stack. We pop the **top two** elements off (19 and 4), subtract them ($19 - 4 = 15$) and push the result:

```
[15]
```

There are tons of other operators in PicoScript, including some just for stack manipulation. For example, you can explicitly `pop` to discard the top element if you don't need it anymore. You can use `exch` to swap the positions of the top two elements, or `dup` to duplicate the top element.

In the following notation, the expression to the left of the percent sign `%` produces the stack shown to the right. (The `%` introduces a line comment, so only the portions to the left are actual PicoScript code.)

```
7 6 5 pop   % [7, 6]

7 6 5 exch  % [7, 5, 6]

7 6 5 dup   % [7, 6, 5, 5]
```

Most languages use stacks to some extent – for example, the **call stack** in C++, Java, or Python. But in a stack-based language (also called a **stack machine**), the stack is the primary way not just to pass parameters to a subroutine but also to store temporary values, replacing many uses of local variables.

PicoScript uses a slash character (`/`) before an identifier to indicate that it's a **symbol.** It should be pushed **as-is** onto the stack, rather than interpreted as an operator. These two examples demonstrate the difference:

```
7 6 5 pop   % [7, 6]

7 6 5 /pop  % [7, 6, 5, /pop]
```

We can use matching curly braces `{ }` to create a **block:** a sequence of tokens that are pushed onto the stack as a group, also without being interpreted:

```
{3 8 add} 5 6 mul   % [{3 8 add}, 30]
```

So the above stack has two items on it, one is a sequence of tokens and the other is an integer. (The `add` operator was part of the block so it was not interpreted, but the `mul` operator was interpreted as usual.)

We use these facilities – symbols and blocks – to define meanings of new operators that are not built-in. For example, PicoScript doesn't have a built-in operator to square a number:

```
6 square  % Error: undefined
```

But we can define one like this:

```
/square {dup mul} def
```

where `def` is a built-in operator that pops a symbol and a block and binds them together in the **symbol table** for future use. Thereafter when you refer to `square` it substitutes the definition. So `6 square` becomes `6 dup mul`. Then we duplicate the number producing `6 6 mul` and finally multiply to get `36`.

Syntax

Tokens in PicoScript consist of integer literals, string literals, symbols, operators, and curly braces. We'll examine each below.

PicoScript supports comments that begin with a percent sign and run to the end of the line. For example, I could document my `square` operator like this:

```
/square {% This operator will square a number
  dup    % First duplicate the number
  mul    % Then multiply the number by itself
} def
```

White space consists of the ASCII space (32), newline (10), carriage return (13), and tab (9) characters. They are interchangeable except within comments and strings.

Integer literals consist of sequences of decimal digits (`0`…`9`) optionally starting with a negative sign (`-`). Unlike in Java or C/C++, integers are unbounded in magnitude.

String literals are delimited by matched parentheses, as in `(Hello  world.)`. The parentheses are just delimiters, not taken as part of the string, so the initial character of that string is `H`. String literals may span across multiple lines, and the resulting string will contain newline characters:

```
(This string
goes across
three lines so contains two newlines.)
```

We use the backslash character to embed special codes such as `\n` for another way to achieve a newline, or `\\` for a literal backslash, or `\(` and `\)` to insert parentheses that do not terminate the string. Parentheses may also be used **without** backslashes as long as they are nested and balanced appropriately:

```
(This (string (contains many) parentheses) up to here ->)
```

But if we want unbalanced parentheses within a string, we must use backslashes:

```
(This \) string is \( fine too.)
```

Identifiers (used for both symbols and operators) must begin with an alphabetic character (upper or lower-case) and thereafter can contain *any characters* except for white-space, curly braces `{}`, parentheses `()`, or the percent sign `%`. Identifiers followed by any of those separator characters do not need additional white-space, so this code consists of the symbol `foo` followed by the string `bar`, rather than a single symbol `foo(bar)`:

```
/foo(bar)
```

And this is the symbol `foo` followed by a comment, rather than `foo%bar`:

```
/foo%bar
```

Unlike in C/C++ or Java, it is acceptable to include any *other* symbols in symbols or operator names:

```
/my2ndScore!
/another*great$day#
```

Running PostScript

Ghostscript is a free interpreter for PostScript that we'll use to illustrate how the language works. You can install it on MacOS X using Homebrew, on Windows using Cygwin, or on GNU/Linux using the native package manager. I also recommend installing `rlwrap` to add command editing to the Ghostscript prompt. Run it like this — the `%` represents your usual shell prompt; the rest is output by the interpreter.

```
% rlwrap gs
GPL Ghostscript 9.18 (2015-10-05)
Copyright (C) 2015 Artifex Software, Inc.  All rights reserved.
This software comes with NO WARRANTY: see the file PUBLIC for details.
GS>
```

To exit, type `quit` at the prompt or hit `^D` (or maybe `^Z` on Windows).

If you see an error like:

```
GPL Ghostscript 9.15: Cannot open X display `(null)'.
```

don't panic — for the most part we won't be using the display capabilities. To run Ghostscript without the display page window, add this option to the command:

```
% rlwrap gs -sDEVICE=bbox
```

However if you do have the display page, let's take a moment to try it out. Type these commands at the `GS>` prompt:

```
GS> /Times-Roman findfont 32 scalefont setfont
GS> (Hello world!) 72 250 moveto show
```

You should see the message appear in the lower left of the display window.

Now you can try some of the examples in the overview section. The prompt will tell you how many items appear on the operand stack, so it's `GS>` when the stack is empty, but changes to `GS<2>` when there are two things on the stack. You can see the contents of the stack using the operator `pstack`; it displays the items top-down:

```
GS> 3 5 mul 7 6 add
GS<2> pstack
13
15
GS<2> add
GS<1> pstack pop
28
GS>
```

Exit the Ghostscript interpreter by pressing control-D (or maybe control-Z).

Library

This section contains a categorized list of essential built-in operators in PicoScript.

Stack operators

These operators are purely for manipulating values on the stack, so they work with values of any type.

pop Discard top element.

```
52 54 pop                    % [52]

/a (b) pop                   % [a]
```

exch Exchange top two elements.

```
52 54 exch                   % [54, 52]

/a (b) exch                  % [b, a]
```

dup Duplicate top element.

```
71 dup                       % [71, 71]

45 50 dup                    % [45, 50, 50]
```

copy Duplicate top N elements, where N is the integer on the top of the stack (and not included in the result).

```
90 80 70 60 2 copy           % [90, 80, 70, 60, 70, 60]

13 14 15 3 copy              % [13, 14, 15, 13, 14, 15]
```

index Duplicate an arbitrary element on the stack. Writing `0 index` is equivalent to dup, but we can use positive integers to dig deeper into the stack.

```
13 14 15 0 index             % [13, 14, 15, 15]

13 14 15 1 index             % [13, 14, 15, 14]

13 14 15 2 index             % [13, 14, 15, 13]
```

clear Discard all elements on the stack.

```
71 72 73 clear               % []

clear                        % []
```

count Push the current size of the stack (not including this result). Also, it leaves all the elements on the stack.

```
71 72 73 count               % [71, 72, 73, 3]

99 count                     % [99, 1]

count                        % [0]
```

Arithmetic operators

These are the usual arithmetic operators on integers. The full Postscript language also supports conversion to floating-point numbers, which is why we distinguish `idiv` for integer division. PicoScript does not support floating-point.

add Add two integers.

```
3 4 add                    % [7]

3 -9 add                   % [-6]
```

idiv Integer division – result is an integer.

```
60 4 idiv                  % [15]

60 3 idiv                  % [20]

60 7 idiv                  % [8]
```

mul Multiply two integers

```
3 4 mul                    % [12]

3 -4 mul                   % [-12]

-3 -4 mul                  % [12]
```

sub Subtract two integers. Be careful about the ordering, because subtraction is not commutative.

```
9 2 sub                    % [7]

2 9 sub                    % [-7]
```

mod Remainder after dividing – like the % operator in C/C++/Java.

```
8 3 mod                    % [2]

60 7 mod                   % [4]
```

String operators

Here are operators on strings and characters. There is not a distinct value type for characters in PicoScript – they are either integers or strings of length 1. In PostScript, strings are explicitly allocated and are mutable; in PicoScript they are immutable. Therefore we support some different operations on them, that will not work in Ghostscript (noted below).

length Return the number of characters in a string or symbol. Fails on values of other types.

```
(Hello) length                  % [5]

/flibbert length                % [8]

() length                       % [0]

(With\)\(parens) length         % [12]

(with\nnew\nlines\n) length     % [15]
```

get Return Nth character in string, as an integer (the ASCII/Unicode value of that character. Indexing starts at zero.

```
(ABCDE) 0 get                   % [65]

(zyxwv) 4 get                   % [118]
```

strcat Concatenate two strings into a new string. This is *not* a built-in operator in Postscript, but we're trying to make Picoscript strings more convenient (and immutable).

```
(Hello) (World) strcat          % [HelloWorld]

(Hello) (!) strcat              % [Hello!]

() (Yes) strcat                 % [Yes]

(Yu) (Gi) (Oh) strcat strcat    % [YuGiOh]
```

tostr Convert integers or Booleans to strings. This is also *not* a built-in operator in Postscript.

```
128 256 mul tostr               % [32768]

128 256 mul tostr length        % [5]

(Beacon) 11 12 add tostr strcat % [Beacon23]
```

tochar Convert an integer to a Unicode character, represented as a string of length 1. This is *not* a built-in operator in Postscript.

```
65 tochar                          % [A]

97 tochar                          % [a]

8364 tochar                        % [€]

76 67 tochar exch tochar strcat    % [CL]
```

Relational operators

Picoscript has distinct Boolean values which can be generated by the operators `true` and `false`.

true Produce the Boolean value *true*.

```
true                               % [true]
```

false Produce the Boolean value *false*.

```
false                              % [false]
```

eq Test whether values are equal. Works on integers, strings, symbols, and Booleans. For any other types (such as blocks), it always returns false – even if the values *appear* to be the same.

```
3 5 eq                             % [false]

3 3 eq                             % [true]

(Hi) (Ho) eq                       % [false]

(Hi) (Hi) eq                       % [true]

/hi /hit eq                        % [false]

/hit /hit eq                       % [true]

true false eq                      % [false]

true true eq                       % [true]

{dup} {dup} eq                     % [false]
```

ne Test whether values are **not** equal. Works on integers, strings, symbols, and Booleans. For any other types (such as blocks), it always returns true – even if the values *appear* to be the same.

```
3 5 ne                          % [true]

3 3 ne                          % [false]

(Hi) (Ho) ne                    % [true]

(Hi) (Hi) ne                    % [false]

/hi /hit ne                     % [true]

/hit /hit ne                    % [false]

true false ne                   % [true]

true true ne                    % [false]

{dup} {dup} ne                  % [true]
```

ge Test greater than or equal. Works on integers or strings. Applying to other types is a run-time error.

```
3 4 ge                          % [false]

4 3 ge                          % [true]

4 4 ge                          % [true]

(Ha) (Heh) ge                   % [false]

(Heh) (Ha) ge                   % [true]

(Ha) (Ha) ge                    % [true]
```

gt Test strictly greater than. Works on integers or strings. Applying to other types is a run-time error.

```
3 4 gt                          % [false]

4 3 gt                          % [true]

4 4 gt                          % [false]

(Ha) (Heh) gt                   % [false]

(Heh) (Ha) gt                   % [true]

(Ha) (Ha) gt                    % [false]
```

le Test less than or equal. Works on integers or strings. Applying to other types is a run-time error.

```
3 4 le                          % [true]

4 3 le                          % [false]

4 4 le                          % [true]

(Ha) (Heh) le                   % [true]

(Heh) (Ha) le                   % [false]

(Ha) (Ha) le                    % [true]
```

lt Test strictly less than. Works on integers or strings. Applying to other types is a run-time error.

```
3 4 lt                          % [true]

4 3 lt                          % [false]

4 4 lt                          % [false]

(Ha) (Heh) lt                   % [true]

(Heh) (Ha) lt                   % [false]

(Ha) (Ha) lt                    % [false]
```

and Logical and operator. Works only on Booleans.

```
false false and                 % [false]

false true and                  % [false]

true false and                  % [false]

true true and                   % [true]
```

or Logical or operator. Works only on Booleans.

```
false false or                  % [false]

false true or                   % [true]

true false or                   % [true]

true true or                    % [true]
```

not Logical not operator. Works only on Booleans.

```
false not                       % [true]

true not                        % [false]
```

Control operators

These operators provide conditional and repeated execution, like the conditionals and loops in most languages. They make heavy use of blocks to defer computation.

if Execute a block if a Boolean value is true.

```
(OK) true {(Yea)} if                % [OK, Yea]

(OK) false {(Yea)} if               % [OK]
```

ifelse Execute one block if true, another block if false.

```
(OK) true {(Yea)} {(Nay)} ifelse    % [OK, Yea]

(OK) false {(Yea)} {(Nay)} ifelse   % [OK, Nay]
```

repeat Execute a block a fixed number of times. The example starts with 1 and doubles it ten times, producing $2^{10} = 1024$.

```
1 10 {2 mul} repeat                 % [1024]
```

for Implement a counting loop that provides its counter on the top of the stack each time it executes the block. The example adds all the odd numbers between 1 and 100: $1 + 3 + 5 + 7 + \ldots + 97 + 99$.

```
0 1 2 100 {add} for                 % [2500]
```

while Takes two blocks – the first is a Boolean-producing expression that tells us whether to continue executing the second block. This is *not* part of Postscript, which instead uses `loop` and `exit` (like `break` in C/C++/Java).

```
1 {dup 1000 lt} {2 mul} while       % [1024]
```