The PostScript programming language[1]

1 History and overview

The primary advantages of the programming language PostScript are that it is simple, efficient, and platform-independent. It is an interpreted language, but the interpreter is simple enough that it does not overburden processors. On the other hand, it is powerful enough to be a genuine programming language — not just a markup language, like HTML.

Today, people use PostScript primarily for describing graphics. But PostScript's programming facilities are interesting enough to make them worth studying on their own. In this paper, we'll study PostScript as a programming language: Our primary interest lies not in describing graphics, but in describing algorithms using PostScript.

1.1 History

PostScript derives historically from the language Forth, retaining many of its features. Its development began in 1976, when John Gaffney developed the language called *Design System* for databases of three-dimensional graphics. John Warnock and Martin Newell picked it up at Xerox PARC in 1978 for their own reasons. And it met its final version — and its name *PostScript* — when John Warnock and Chuck Geschke founded Adobe Systems. Adobe continues to manage PostScript and retains a trademark on its name.

Today, PostScript's primary use is for describing documents — for storing paper documents electronically and for sending documents to printers. Its simplicity, efficiency, and platform-independence make it ideal for the wide variety of printers with moderate processing capacity.

1.2 The stack

PostScript is built entirely around the concept of **the stack**. (To be specific, it's the **operand stack**, but it's so important that the *operand* part of the name is often omitted.) The stack holds the data that the program is currently using, including parameters and return values, intermediate results, and even program code.

Throughout this tutorial, I'll draw the stack with the topmost element on the right.

1.3 Data

The stack can hold a variety of types of information. The following enumerates the types that are important in our study.

- numeric objects (integers, reals, and Booleans)
- strings and arrays
- names (A **name** is any sequence of non-whitespace, non-special characters that can't be understood as a number. PostScript's **special characters** are '`(`', '`)`', '`<`', '`>`', '`[`', '`]`', '`{`' '`}`', '`%`', and '`/`'.)
- dictionaries (We'll see these in Section 4.)

Associated with each piece of data is a single bit called the **executable attribute**. If this bit isn't set, it is said to have the **literal attribute**; executing a literal piece of data simply pushes the object onto the stack. If it has the executable attribute, however, executing the data may have different behavior.

- An executable integer, real, or string simply pushes itself onto the stack.
- An executable name executes the operation associated with the name.
- An executable array executes each item in the array, in sequence.

A program is just a sequence of pieces of data, separated by whitespace. The PostScript interpreter executes each piece of data in sequence.

2 Programming

That's the abstract, introductory stuff. Now let's delve into some actual programs. We'll begin with a simple program to multiply two numbers together, to illustrate how the stack works. Then we'll look at a program that uses some of PostScript's control structures. Then we'll see how a PostScript program can define variables and procedures. We'll tackle a recursive PostScript function, and close with some stylistic comments.

2.1 A simple program

Let's start with a very simple PostScript program.

```
6 9 mul =
```

When the interpreter is given this program, it prints "54" to the display. In fact, you can try this out on a computer by using a popular PostScript interpreter called *Ghostscript.* On many Unix systems, you can invoke it as follows.

```
% gs              Type ''gs'' at the prompt to start it
Aladdin Ghostscript 3.33 (4/10/1995)
Copyright (C) 1995 Aladdin Enterprises, Menlo Park, CA.  All rights
reserved.
This software comes with NO WARRANTY: see the file COPYING for details.
GS>6 9 mul =      ''GS>'' is the prompt. Type the program.
54
GS>                Ghostscript is prompting for more.
```

To stop Ghostscript, type control-C.

Here's how our PostScript program works.

1. When the PostScript interpreter reads a number, it pushes the number onto the stack. So the first word, "6," being a number, gets pushed onto the stack.
2. The next word, "9," gets pushed onto the stack. The stack now holds `6 9`.
3. When the interpreters encounters a name like `mul`, it executes what is associated with the name. In this case, it pops the top two items off the stack, multiplies them together, and pushes the product onto the stack again. (If the top two items aren't both numeric, the interpreter would give a type error.) So the stack is now `54`.
4. Then the interpreter encounters the name `=`, which pops the top piece of data from the stack and prints it to the screen. At this point the interpreter prints "54" to the screen.

2.2 Control structures

PostScript provides a variety of names that provide useful ways of controlling execution of other pieces of data. For example, the `repeat` name pops two items from the stack: first an executable array, and then an integer. (If the first isn't an executable array, or the second isn't an integer, there is a type error.) Then `repeat` executes the executable array the number of times described by the integer.

For example, the following program finds 5^3.

```
1 3 { 5 mul } repeat =
```

The braces specify an array with the executable attribute. When the interpreter encounters this, it pushes the entire executable array onto the stack.

If run, the interpreter prints "125". Here is what happens.

1. The interpreter pushes 1. Stack: `1`.
2. The interpreter pushes 3. Stack: `1 3`.
3. The interpreter pushes the executable array `{5 mul}`. Stack: `1 3 {5 mul}`.
4. The interpreter executes `repeat`. Both 3 and `{5 mul}` are popped. Stack: `1`.
   (a) The interpreter executes `{5 mul}` once. First 5 is pushed onto the stack and then `mul` is executed. Stack: `5`.
   (b) The interpreter executes `{5 mul}` a second time. First 5 is pushed onto the stack and then `mul` is executed. Stack: `25`.
   (c) The interpreter executes `{5 mul}` the third time. First 5 is pushed onto the stack and then `mul` is executed. Stack: `125`.
5. The interpreter executes `=`. The interpreter pops 125 and displays it.

The `if` name gives you a way of specifying whether something should occur. It pops off first an executable array and then a Boolean. If the Boolean is *true*, it executes the executable array. Otherwise, nothing more happens. An example:

```
3 0 gt { 42 = } if
```

The `gt` name determines pops off two numbers and pushes the Boolean value *true* if the second number popped is greater than the first, and *false* otherwise. So this sequence prints 42 if $3 > 0$, and nothing otherwise. (In other words, it just prints 42 — I didn't claim it was a very interesting example.)

2.3 Associating data with names

The `def` name gives a way of associating a piece of data with a name. The `def` procedure pops a piece of data, and then a name from the stack. It then associates the piece of data with the name. In this way, you can have something like a variable or a procedure.

But so far, we don't have a way of getting a name onto the stack — every time the interpreter encounters a name, it just executes it. We can get a name onto the stack by prefixing it with a slash. (The slash gives the name the literal attribute.)

```
/pi 3.14159 def
```

The above example associates the name `pi` with the value 3.14159. So later we can use `pi` if we like.

```
6 pi mul =
```

When the interpreter reaches `pi`, it executes the value of `pi`; as a number, this value pushes itself onto the stack. The net effect of the above PostScript is to display the number 18.8495.

By associating a name with an executable array, you can get a procedure or function. For example, the following defines a function that computes the circumference of a circle given its radius. (Recall $C = 2\pi r$.)

```
/circumference { 2 pi mul mul } def
3 circumference =
```

When given to the interpreter, the data `{2 pi mul mul}` is associated with the name `circumference`. The `3` is then pushed onto the stack, and then the executable array associated with `circumference` is executed.

2.4 Recursion

Before looking at this final program, there is one final category of useful operators: stack operators. One of the simplest is `dup`, which pops off one data item from the stack and pushes two copies of it back on.

Now consider the following recursive program to compute the factorial of a number. (The `sub` name pops the top two numbers from the stack and pushes the result of subtracting them.)

```
/fact { dup 1 gt { dup 1 sub fact mul } if } def
4 fact =
```

As you might expect, the above would print 24.

Figure 1 illustrates how the stack changes over time as the `fact` name is executed in this example. As you will see if you try to step through this yourself, PostScript maintains its own **execution stack** internally, just as other programming languages. The operand stack is independent of the execution stack.

2.5 Programming style

Comments in PostScript begin with a percent sign (‘`%`’) and extend to the end of the line.

We think left-to-right, but PostScript control structures are inverted from this. For example, the `if` word doesn’t come until *after* we list what should be done if the condition is true. When you use `if`, `repeat`, or another control structures, it’s good programming style to use a comment at the beginning of each executable array to tell the gentle reader what is coming after the executable array.

Here is our earlier recursive factorial program, formatted more nicely.

```
/fact { %def
    dup 1 gt { %if
        dup 1 sub fact mul
    } if
} def

4 fact =                        % compute and print 4!
```

3 PostScript operators

Mastering PostScript programming from here is largely a matter of broadening your knowledge of the library of names built into the language. Here is a sampling of the most important ones for the purposes of programming.

```
4             Execution begins with 4 previously pushed.
4 4
4 4 1
4 true
4 true {dup 1 sub fact mul}
4 4           We begin executing the array
4 4 1
4 3
4 3 3         We've entered the recursive call, duplicating
4 3 3 1
4 3 true
4 3 true {dup 1 sub fact mul}
4 3 3
4 3 3 1
4 3 2
4 3 2 2       We've entered another recursive call
4 3 2 2 1
4 3 2 true
4 3 2 true {dup 1 sub fact mul}
4 3 2 2
4 3 2 2 1
4 3 2 1
4 3 2 1 1     We've entered another recursive call
4 3 2 1 1 1
4 3 2 1 false
4 3 2 1 false {dup 1 sub fact mul}
4 3 2 1       We're about to exit this recursive call
4 3 2         We exit another recursive call
4 6           We exit another recursive call
24            We exit the initial execution of fact
```

Figure 1: Stack content changes as `4 fact` executes.

3.1 Stack operators

clear Pops everything from the stack.

dup Duplicates the top element of the stack.

exch Exchanges the top two elements in the stack.

index Pops an integer n from the stack and pushes a copy of the stack's nth object (with 0 being the object that was just below n).

pop Pops the top element from the stack.

stack Prints the contents of the stack (nothing is removed).

= Pops the top element from the stack and prints it.

3.2 Arithmetic operators

add Pops the top two numbers and pushes their sum.

div Pops the top two numbers and pushes their quotient as a real.

eq Pops the top two numbers and pushes a Boolean saying whether they are equal.

ge Pops the top two numbers and pushes a Boolean saying whether the second is at least the top.

gt Pops the top two numbers and pushes a Boolean saying whether the second is greater than the top.

idiv Pops the top two numbers and pushes their integer quotient ignoring any remainder.

le Pops the top two numbers and pushes a Boolean saying whether the second is at most the top.

lt Pops the top two numbers and pushes a Boolean saying whether the second is less than the top.

mod Pops the top two numbers and pushes their modulus (remainder).

mul Pops the top two numbers and pushes their product.

ne Pops the top two numbers and pushes a Boolean saying whether they are not equal.

sub Pops the top two numbers and pushes their difference.

3.3 Logical operators

and Pops the top two Booleans and pushes the logical AND of them.

false Pushes the Boolean value *false*

not Pops the top Booleans and pushes the logical NOT of it.

or Pops the top two Booleans and pushes the logical OR of them.

true Pushes the Boolean value *true*

3.4 Control operators

exec Pops the top element of the stack and executes it.

exit Exits the currently executing loop.

for Pops four items: the integer $init$, the integer $incr$, the integer $final$, and an executable array. Then, for each integer starting at $init$ and going by steps of $incr$ until passing $final$, it pushes the current integer onto the stack and then executes the array.

The following example is an alternative way of computing $4!$.

```
1 1 1 4 { mul } for
```

if Pops a Boolean and an executable array from the stack. If the Boolean is *true*, the interpreter executes the array.

ifelse Pops a Boolean and two executable arrays from the stack. The interpreter executes the first array if the Boolean is *true*, and the second (top) array otherwise.

loop Pops an executable array from the stack and repeatedly executes it (until `exit` is executed).

repeat Pops an integer and an executable array and executes the array the number of times given by the integer.

3.5 Graphics operators

PostScript graphics maintain a **path** as you draw. Conceptually, PostScript maintains a cursor location on the page. As you move, you accumulate a path. The path is not displayed until you execute the `stroke` operator.

closepath Closes the path by adding a line from the current position to the path's starting point.

lineto Pops the top two integers x and y and adds a line from the current position to (x, y) to the path.

moveto Pops the top two integers x and y and starts a new path from the position (x, y).

showpage Shows the current page (printing it on a printer) and then clears the graphics buffer for a new blank page.

stroke Draws the current path as accumulated so far.

4 Dictionaries

If you're a narrow-minded programmer whose mind is yet to be broadened by PostScript, you might be tempted define `fact` the following way.

```
/fact {
    /i exch def
    1
    1 1 i { mul } for
} def
```

This uses a more traditional imperative style by defining the name `i` to be the parameter and then using it later. This `fact` definition does indeed work.[2]

However, it has a problem. Suppose `fact` is being used within a larger context that has its own designs on the `i` name.

```
0
1 1 5 { /i exch def
        5 fact
        5 i sub fact idiv
        i fact idiv
} for
```

[2]This is somewhat, contrived, though. A simpler definition is the following: `/fact {1 exch -1 1 {mul} for} def`.

This program won't work as surely was intended, because each time `fact` occurs, `i` gets *redefined* with a different value.

PostScript provides a solution to this quandary: the **dictionary**. The dictionary is a different type of data that represents a namespace. The `def` operator associates a value with a name in the current dictionary, but with other operators you can change the current dictionary.

In fact, there is yet another stack called the **dictionary stack**. The interpreter always uses the top dictionary in the dictionary stack when it encounters a name, but as a program progresses you can push and pop dictionaries from the dictionary stack.

Dictionaries give the programmer the opportunity to define a separate namespace into which definitions go. Here is a modified `fact` that illustrates the use of a dictionary.

```
/fact {
    1 dict begin
    /i exch def
    1
    1 1 i { mul } for
    end
} def
```

Here are the dictionary-related operators.

begin Pops a dictionary from the operand stack and pushes it to the dictionary stack.

currentdict Creates a copy of the top of the dictionary stack and places it in the operand stack.

def Pops a name and a piece of data from the stack and adds (or replaces) an association of the name with the data into the current dictionary.

dict Pops an integer from the operand stack and pushes a blank dictionary to the operand stack. The maximum number of names defined in the dictionary is given by the popped integer.

end Pops the top of the dictionary stack.

5 Printable PostScript

If you have a PostScript program that uses PostScript's graphical capabilities, you may want to send it to a printer. Printers generally require the following as the *first* line if the PostScript program.

```
%!PS-Adobe-2.0
```

For example, if you made the following text file and printed it to a PostScript printer, it would print a page with a diagonal line across it.

```
%!PS-Adobe-2.0
0 0 moveto 500 500 lineto stroke showpage
```