POSTSCRIPT

LANGUAGE

SUPPLEMENT

ADOBE SYSTEMS
INCORPORATED

POSTSCRIPT Language Supplement

January 9, 1986

1 POSTSCRIPT LANGUAGE CHANGES

Several additions have been made to the standard POSTSCRIPT language. These additions are upward-compatible and do not affect the function of any existing POSTSCRIPT programs. The changes are included in all POSTSCRIPT implementations with version number 25.0 or higher; they will be documented in a future edition of the *POSTSCRIPT Language Reference Manual*.

In general, POSTSCRIPT programs that are intended to be compatible with all POSTSCRIPT printers should not make use of the new features. However, it is possible for a program to determine whether or not the new features are present and to invoke them conditionally. The descriptions below suggest how to determine whether a particular feature is present or absent.

Packed arrays

POSTSCRIPT procedures are represented as executable arrays which, until now, have been stored in the same fashion as literal data arrays. This representation, while offering maximum flexibility, is very costly in space (8 bytes per element). Large POSTSCRIPT programs, such as the built-in server program and downloaded preambles, consume considerable amounts of VM.

Since most programs do not require the ability to be treated as data but only the ability to be executed, a more compact representation has been introduced: the *packed array*. Programs represented as packed arrays are typically 50 to 75 percent smaller than the same programs represented as ordinary arrays.

A packed array object has a type different from an ordinary array object (‘packedarraytype’ versus ‘arraytype’); but in most respects it behaves the same as an ordinary array. You can execute a packed array; you can extract elements (using **get**) or subarrays (using **getinterval**); you can enumerate it (using **forall**); and so forth. Individual elements extracted from a packed array are ordinary POSTSCRIPT objects; a subarray of a packed array is also a packed array.

The differences between packed arrays and ordinary arrays are:

- Packed arrays are always read-only: you can’t use **put**, **putinterval**, etc., to store into one.
- Packed arrays are created differently from ordinary arrays (see below).
- Accessing arbitrary elements of a packed array can be quite slow; however, accessing the elements sequentially (as is done by the POSTSCRIPT interpreter and by the **forall** operator) is approximately as efficient as accessing an ordinary array.
- The **copy** operator cannot copy into a packed array (since it is read-only); however, it can copy the value of a packed array to an ordinary array of at least the packed array’s length.

There are two ways in which packed arrays come into existence. The first and more common way is for the POSTSCRIPT input scanner to create packed arrays automatically for all executable arrays that it reads. That is, whenever the scanner encounters a ‘{’ while reading a file or string, it accumulates all tokens up to the matching ‘}’ and turns them into a packed array instead of an ordinary array.

The choice of array type is controlled by a mode setting, manipulated by the new operators **setpacking** and **currentpacking** (described at the end of this section). If the array packing mode is *true*, POSTSCRIPT procedures encountered subsequently by the scanner are created as packed arrays; if the mode is *false*, procedures are created as ordinary arrays. The default value is *false* (i.e., create ordinary arrays), for compatibility with existing programs.

The other way to create a packed array is to build it explicitly by invoking the **packedarray** operator with a list of operands to be incorporated into a new packed array.

Immediately evaluated names

The language syntax has been extended to include a new kind of name token, the *immediately evaluated name*. When the scanner encounters the token ‘//*name*’ (a name preceded by two slashes with no intervening spaces), it immediately looks up the name in the context of the current dictionary stack and substitutes the corresponding value for the name. If the name is not found, an **undefined** error occurs.

The substitution occurs *immediately*, regardless of whether or not the token appears inside an executable array delimited by ‘{...}’. Note that this process is a substitution and not an execution; that is, the name’s value is not executed but rather is substituted for the name itself, just as if the **load** operator had been applied to the name. This action is related to the action performed by the **bind** operator (see the *PostScript Language Reference Manual*); but whereas **bind** performs substitution only for names whose values are operators, each occurrence of the ‘//*name*’ syntax is replaced by the value associated with *name* regardless of the value’s type. The following examples illustrate this:

```
/a 3 def
/b {(test) print} def
//a ⇒ 3
//b ⇒ {(test) print}
{//a //b a /b} ⇒ {3 {(test) print} a /b}
```

The purpose of using immediately evaluated names is similar to that of using the **bind** operator: to cause names in procedures to become ‘tightly bound’ to their values. However, a word of caution is in order: indiscriminate use of immediately evaluated names may change the semantics of a program. In particular, recall that when the interpreter encounters a procedure object *directly* it simply pushes it on the operand stack; but when it encounters a procedure object *indirectly* (by looking up an executable name) it executes the procedure. (See section 3.6 of the *PostScript Language Reference Manual*.) Therefore, execution of the program fragments:

```
{... b ...}
{... //b ...}
```

may have different effects if the value of the name ‘`b`’ is a procedure.

The immediately evaluated name facility is present in all versions of the PostScript interpreter since version 25.0 (as reported by the **version** operator). Earlier versions of the interpreter will scan ‘//*name*’ as two distinct tokens: ‘/’, a literal name with no text at all, and ‘/*name*’, a literal name whose text is *name*.

Font cache operation

Operation of the font cache has changed somewhat. Formerly there was a single limit on the number of bytes occupied by a character in the cache; a character larger than that would not be cached. Now there are two cache thresholds, a lower and an upper one. If a character is larger than the upper threshold (as determined by the bounding box specified to **setcachedevice**), it will not be cached; otherwise it will be. If it is cached and is larger than the lower threshold, it will be *compressed*; otherwise it will be stored as a full pixel array.

The two thresholds are manipulated by the new operators **setcacheparams** and **currentcacheparams**, described below. The old operators, principally **cachestatus** and **setcachelimit**, remain valid. It is a rare POSTSCRIPT program that needs to deal with these operators.

Compressed characters consume much less space in the font cache than full pixel arrays (by factors of up to 40), but require more computation to reconstitute when they are needed. Reconstituting a compressed character is still substantially faster than re-executing the original character description. In systems printing at 300 pixels per inch or less, the default lower threshold is set so that characters up to about 20 points are stored as full pixel arrays while larger ones are stored in compressed form. This causes ordinary body text to be cached using the time-efficient full pixel array representation but large characters to be cached using the space-efficient compressed representation.

New Operators

setpacking

bool **setpacking** –

sets the array packing mode to the specified boolean value. This determines the type of executable arrays subsequently created by the POSTSCRIPT scanner. The value *true* selects packed arrays; *false* selects ordinary arrays.

The packing mode affects only the creation of procedures by the scanner when it encounters program text bracketed by ‘{’ and ‘}’ during interpretation of an executable file or string object or during execution of the **token** operator. It does not affect the creation of literal arrays by the ‘[’ and ‘]’ operators or by the **array** operator.

The array packing mode setting persists until overridden by another execution of **setpacking** or until undone by a **restore**.

EXAMPLE:

```
systemdict /setpacking known
   {/savepacking currentpacking def
    true setpacking
   } if

... arbitrary procedure definitions ...

systemdict /setpacking known {savepacking setpacking} if
```

If the packed array facility is available, the procedures represented by ‘arbitrary procedure definitions’ are defined as packed arrays; otherwise they are defined as ordinary arrays. This example is careful to preserve the array packing mode in effect before its execution.

ERRORS:
stackunderflow, typecheck

currentpacking

– **currentpacking** bool

returns the array packing mode currently in effect.

STANDARD VALUE: false

ERRORS:
stackoverflow

packedarray any_0 ... any_{n-1} n **packedarray** packedarray

creates a packed array object of length *n* containing the objects any_0 through any_{n-1} as elements. **packedarray** first removes the non-negative integer *n* from the operand stack. It then removes that number of objects from the operand stack, creates a packed array containing those objects as elements, and finally pushes the resulting packed array object on the operand stack.

The resulting object has a type of ‘packedarraytype’, a literal attribute, and read-only access. In all other respects, its behavior is identical to that of an ordinary array object.

STANDARD VALUE: false

ERRORS:
rangecheck, stackunderflow, typecheck, VMerror

setcacheparams mark lower upper **setcacheparams** –

sets cache parameters as specified by the integer objects above the topmost mark on the stack, then removes all operands and the mark object as if by **cleartomark**.

The number of cache parameters is variable.[1] If more operands are supplied to **setcacheparams** than are needed, the topmost ones are used and the remainder ignored; if fewer are supplied than are needed, **setcacheparams** implicitly inserts default values between the mark and the first supplied operand.

The *upper* operand specifies the maximum number of bytes that may be occupied by the pixel array of a single cached character, as determined from the information presented by the **setcachedevice** operator. This is the same parameter as is set by **setcachelimit**; see the description of that operator in the *POSTSCRIPT Language Reference Manual*.

The *lower* operand specifies the threshold at which characters are stored in compressed form rather than as full pixel arrays. If a character's pixel array requires more than *lower* bytes to represent, it will be compressed in the cache and reconstituted from the compressed representation each time it is needed.

Setting *lower* to zero forces all characters to be compressed, permitting more characters to be stored in the cache but increasing the work required to print them. Setting *lower* to a value greater than or equal to *upper* disables compression altogether.

ERRORS:
rangecheck, unmatchedmark

currentcacheparams – **currentcacheparams** mark lower upper

pushes a mark object followed by the current cache parameters on the operand stack. The number of cache parameters returned is variable (see **setcacheparams**).

STANDARD VALUE: mark 1250 12500

ERRORS:
stackoverflow

---

[1] In future versions of the POSTSCRIPT interpreter there may be more than two cache parameters defined.

showpage and **copypage**

The correct use of **showpage** versus **copypage** is a matter requiring some clarification. Inappropriate use of **copypage** can result in significant performance degradation in new POSTSCRIPT printers.

showpage is the normal operator for causing pages to be output. It has three effects: it prints the current page, it erases the current page, and it reinitializes the graphics state.

copypage is a somewhat more specialized operator that just prints the current page but does not erase it or reset the graphics state. Its main intended use is to permit adding new marks to an existing page, e.g., when building up a page incrementally.

showpage is logically equivalent to the sequence:

```
copypage erasepage initgraphics
```

However, use of **copypage** for printing pages can degrade page throughput significantly. One reason for this is that **showpage** performs the printing and the erasing in parallel whereas the **copypage erasepage** method performs them serially; there are other reasons as well.

copypage should also not be used to defeat the automatic **initgraphics** of **showpage**.[2] That is, to print and erase the current page but leave the graphics state unchanged, you should *not* say:

```
copypage erasepage
```

Instead you should say:

```
gsave showpage grestore
```

Please also note that the correct way to print multiple copies of a page is to associate the desired number of copies with the name **#copies** prior to invoking **showpage**, as discussed under **showpage** in the *POSTSCRIPT Language Reference Manual*. The **#copies** convention now applies uniformly to both **showpage** and **copypage**, whereas formerly it applied only to **showpage**.

[2] Unfortunately, the current *POSTSCRIPT Language Tutorial and Cookbook* includes an example that uses this technique.

Index

Contents