

PostScript™-Sprache

Frank Richter

27.01.2003

Stack-Operationen

`exch` vertauscht die zwei obersten Stackelemente

`x y exch => y x`

`dup` dupliziert oberstes Stackelement

`x dup => x x`

`pop` löscht oberstes Element vom Stack

`x y pop => x`

`index` Kopie eines beliebigen, weiter unten liegenden Elementes

`x y z 2 index => x y z x`

`roll` das oberste Element legt fest, wie oft und in welche Richtung gerollt werden soll. Das zweitoberste bestimmt die Anzahl.

`1 2 3 4 5 3 2 roll => 1 2 4 5 3`

Variablen

- Definition einer Variablen entspricht Zuweisung eines Namens für einen Wert
- Zuerst Namen auf den Stack legen: Slash voranstellen, dann zuzuweisender Wert, anschließend `def`
Bsp: `/ppi 72 def`
- Inhalt einer Variablen läßt sich nur ändern, indem diese neu definiert wird:
Bsp: `/ppi 100 def`
oder: `/ppi ppi 28 add def`
- Zugriff auf Inhalt der Variable ist einfach: der Variablenname kann überall einfach statt des Wertes verwendet werden.
- Typen: integer, real, string, array, dictionary, mark
Typbestimmung zur Laufzeit mit Operator `type`

Strings

- Strings werden in runde Klammern eingeschlossen:
(Eine Testzeichenkette)
- Klammern sind durch vorangestellten Backslash zu maskieren:
(Test in \ (Klammern\))
- Backslashes sind durch Verdoppelung zu maskieren:
(Ein Text mit \\backslashes\\)
- leere String-Objekte lassen sich mit dem `string`-Operator erzeugen.
- Andere Objekte lassen sich mit dem Operator `(cvs)` in einen String konvertieren:
(1+2=) show 1 2 add 10 string cvs show

Arrays

- sind eine geordnete Menge von Werten (möglicherweise unterschiedlichen Typs)
- numerische Indizes, beginnend bei 0
- leeres Array anlegen: `n array => [null null null]`
- initialisiertes Array: `[1 2 3] => [1 2 3]`
- Zugriff auf ein Element: `[1 2 3] 1 get => 2`
- Elemente vom Stack in array: `x y 2 array astore => [x y]`
- array-Elemente auf Stack legen: `[x y] aload => x y [x y]`

Dictionaries

- Eine Tabelle mit Schlüssel-Wert-Paaren, vergleichbar einem Tcl-Array oder einem perl-Hash
- 2 dictionaries sind immer vorhanden:
 - system dictionary** enthält die vordefinierten PostScript™-Operatoren
 - user dictionary** enthält programmdefinierte Variable und Befehle
- es gibt einen *dictionary stack* mit dem *system dictionary* ganz unten, darauf dem *user dictionary* und wiederum darauf programmdefinierten dictionaries
- Namen werden im *dictionary stack* von oben nach unten gesucht.

Prozeduren

- selbst definierte Prozeduren stehen in einem dictionary, zusammen mit Variablen
- werden genau wie Variable definiert, wobei der body der Prozedur in geschweifte Klammern einzuschließen ist.

Bsp: `/mm {72 mul 25.4 div} bind def`

- Der `bind`-Operator bewirkt, dass Operatornamen innerhalb der Prozedur durch die entsprechenden Operationen ersetzt werden. Das bringt eine höhere Geschwindigkeit.
- Eigene Prozeduren werden genau so wie eingebaute Operatoren verwendet:
Bsp: `20 mm 20 mm moveto`

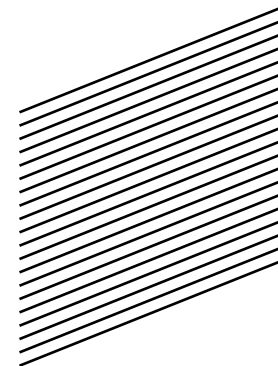
Verzweigungen

- einfache if-Verzweigung: `boolean procedure if => -`
Bsp: `x 1 eq {(x ist 1) =} if`
- mit else-Zweig: `boolean proc1 proc2 ifelse => -`
Bsp: `x 1 eq {(x=1) =} {(x!=1) =} ifelse`
- für den Test gibt es eine ganze Reihe Operatoren, die `true` oder `false` zurückliefern, z.B. `eq`, `ne`, `gt`, `lt`, `ge`, `le`, `and`, `or`, `not`

repeat-Schleife

Die repeat-Schleife: `int proc => -`
Die Prozedur wird einfach n-mal wiederholt.

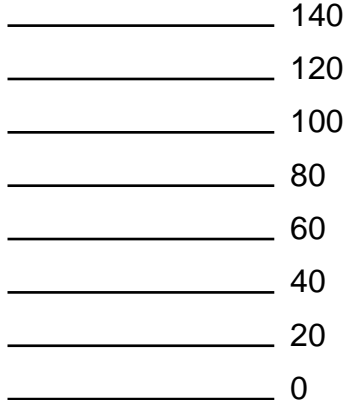
```
%!PS
%%BoundingBox: 0 0 101 136
0 0 moveto
20 { %repeat
  gsave
    100 40 rlineto
    stroke
  grestore
  0 5 rmoveto
} repeat
```



for-Schleife

Die for-Schleife: `initial increment limit proc =>` -
Eine Laufvariable wird von `initial` bis `limit` um jeweils `increment` erhöht und dabei jeweils die Prozedur `proc` ausgeführt. Die Laufvariable steht innerhalb der Prozedur auf dem Stack zur Verfügung.

```
%!PS
%%BoundingBox: 0 0 126 149
/Helvetica findfont 12 scalefont setfont
0 20 140 { %for
  dup 0 exch moveto
  100 0 rlineto stroke
  dup 106 exch moveto
  10 string cvs show
} for
```



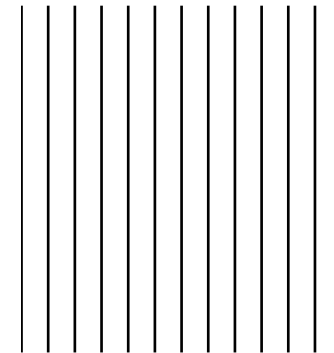
_____	140
_____	120
_____	100
_____	80
_____	60
_____	40
_____	20
_____	0

loop-Schleife

Die loop-Schleife: `proc loop => -`

Die Prozedur `proc` wird endlos wiederholt. Man braucht den `exit`-Operator innerhalb der Prozedur, um die Schleife zu verlassen.

```
%!PS
%%BoundingBox: 0 0 121 131
/x 0 def
{ %loop
  x 120 gt {exit} if
  x 0 moveto
  x 130 lineto stroke
  /x x 10 add def
} loop
```



forall-Schleife

Die forall-Schleife iteriert über die Elemente eines Arrays oder eines Strings:

`array proc forall => -` bzw. `string proc forall => -`

Die Prozedur `proc` wird für jedes Element des Arrays bzw. jedes Zeichen des Strings wiederholt, wobei das Element innerhalb der Prozedur auf dem Stack verfügbar ist.

```
%!PS
%%BoundingBox: 0 23 125 134
/Helvetica findfont 18 scalefont setfont
0 120 moveto
(Gespreizt) { %forall
  1 string dup 0 3 index put
  gsave show grestore
  15 -12 rmoveto
} forall
```

G
e
s
p
r
e
i
z
t

Beispiel - ein Zifferblatt

```
%!PS
%%BoundingBox: 0 0 201 201
100 100 translate
0 0 100 0 360 arc % Kreis
1 setlinewidth stroke

/Helvetica findfont 25 scalefont setfont
1 1 60 { % Beginn der Schleife
  /i exch def
  -6 rotate
  i 5 mod 0 eq { % i durch 5 teilbar?
    i 5 idiv =string cvs % Ziffer als String
    dup stringwidth pop 2 div neg 80 moveto % zentrieren
    show
  } {
    0 100 moveto 0 5 neg rlineto stroke % Minutenstrich
  } ifelse
} for % Ende der Schleife
```

