CREM

Le langage PostScript

1 Introduction

Le langage de programmation `PostScript` est un langage de description de page qui manipule des objets graphiques à l'aide d'opérateurs. Il a été développé par la société américaine Adobe Systems Inc. (John Warnock et Charles Geschke, 1982).

L'idée est de décrire la page graphique pixel par pixel dans une mémoire du périphérique qui va l'imprimer. L'intérêt de travailler en `PostScript` est qu'il donne accès à l'ensemble de toutes les possibilités du périphérique dont on dispose. De plus, le langage est totalement indépendant du périphérique et de l'ordinateur utilisés : la page réalisée est visible aussi bien sur Mac que sur PC ; elle est imprimable aussi bien à partir d'un Mac que d'un PC.

S'il est vrai qu'il existe des imprimantes dites « imprimantes `PostScript` » qui intègrent un interpréteur `PostScript`, il n'est pas nécessaire d'en disposer pour autant. On écrit les commandes du langage avec n'importe quel éditeur de texte, par exemple `Alpha` sur Macintosh et `Bloc-notes` ou `WordPad` sous Windows. Ces commandes sont alors interprétées avec un interpréteur `PostScript` qui permet l'impression au moyen de n'importe quelle imprimante. Sur Macintosh, un tel interpréteur est, par exemple, `MacGSView (2.0b3)` que l'on télécharge gratuitement sur le site Internet

`<http://www.cs.wisc.edu/~ghost/macos/index.htm>`

et, sous Windows, on utilise `GSView` téléchargeable tout aussi gratuitement à l'adresse

`<http://www.cs.wisc.edu/~ghost/gsview/index.html>`.

Sous Windows, une façon bien agréable de travailler est d'ouvrir une demi-fenêtre `Bloc-notes` ou `WordPad` et une demi-fenêtre `GSView`. On introduit alors les commandes dans l'éditeur de texte ; on sauve le fichier puis on l'ouvre dans l'interpréteur `PostScript`, ce qui permet de visualiser tout de suite le travail accompli.

Il reste à installer le logiciel après avoir téléchargé (pour PC)

`gs703w32.exe` (**AFPL Ghostscript**) et `gsv41w32.exe` for `Win32`,

La version de `PostScript` est définitive et unique. Le langage, tel que l'a décrit la société Adobe Systems, est consigné dans l'ouvrage de référence :

PostScript Language
Reference manual
Adobe Systems Inc.
Addison-Wesley Publishing Company.

2 Pour en savoir un peu plus

Le principe du langage, nous l'avons dit, consiste à décrire la page pixel par pixel dans la mémoire image du périphérique. Le transfert de cette page sur le papier s'effectue à l'aide d'un « moteur de marquage » qui associe à un bit de mémoire image une tache sur le papier.

Le langage n'est limité que par les possibilités du périphérique et du moteur de marquage. Il est totalement indépendant du périphérique et de l'ordinateur utilisés. La résolution finale peut aussi bien être de 300 points au pouce ou de 1 270 ou de... Quant à `PostScript`, il a une résolution de 72 points au pouce, ce qui signifie que deux points adressés par le langage sont au minimum distants de $1/72^{\text{ème}}$ de pouce c'est-à-dire un peu plus que 35 centièmes de millimètre.

La plupart des moteurs de marquage utilisent la notion d'impression laser qui s'apparente très fort à la photocopie : les deux techniques utilisent le procédé de xérographie. Un rouleau, qui supporte le papier, est recouvert d'une substance possédant la propriété d'inverser la polarité électrostatique de zones « irradiées » par un pinceau lumineux qui, en l'occurence est un laser à semi-conducteur. Le toner – poudre généralement de couleur noire – se dépose sur les zones déchargées car il est chargé d'une polarité inverse de celle des zones. Il est alors fixé par la chaleur en passant sur un rouleau brûlant.

Une imprimante à laser possède souvent une base de photocopieur améliorée par une électronique qui en fait un ordinateur. Elle possède un microprocesseur, une mémoire vive (RAM) et une mémoire morte (ROM) qui, toutes deux, sont de l'ordre du méga-octet. Une telle quantité de mémoire est nécessaire. En effet, toute imprimante à laser possède une résolution minimum de 300 points par pouce. Ainsi, une page de format A4 – correspondant à 8,5 pouces par 11 pouces – possède une surface de 93,5 pouces carrés, ce qui représente 8 415 000 points ($93,5 \times 300 \times 300$). Si on représente en mémoire cet ensemble en associant un bit à 1 pour un grain de toner déposé et un bit à 0 dans le cas contraire, nous obtenons quelque huit millions de bits c'est-à-dire un méga-octet, puisqu'un octet vaut huit bits. Et cela uniquement pour représenter une page de mémoire !

La mémoire morte, elle, contient l'implémentation de l'interpréteur du langage et la description des différentes polices de caractères ou *fonts*.

John Warnock est à Adobe Systems ce que Bill Gates est à Microsoft. Il est né en 1940, diplômé de l'université d'Utah (maîtrise en mathématiques et doctorat en informatique). Dès 1972, il aborde les problèmes d'association entre un écran de visualisation et une mémoire d'image, chez Evans et Sutherland. Cette société développe des simulateurs de vol et d'installations portuaires. En 1978, il rejoint le PARC (*Palo Alto Research Center*) de Xerox Corporation. Martin Newell et lui-même développent un langage de description de page qu'ils appellent JAM (pour John And Martin), qui donnera naissance au langage Interpress. En 1982, John Warnock et Charles Geschke développent `PostScript` et créent Adobe Systems Inc. à Pala Alto en Californie.

3 Premiers pas

3.1 Page PostScript A4

La résolution du langage étant de 72 points au pouce, les dimensions – en points – d'une page A4 ($8,5 \times 11$ pouces) sont donc 612×792. Le point origine, de coordonnées $(0,0)$ est situé au coin inférieur gauche de la page.

Il faut noter que les imprimantes ne sont pas capables d'écrire sur toute la surface de la page A4 et qu'il y a ainsi une bordure sur chacun des quatre côtés de la page que le moteur de marquage ne peut atteindre.

3.2 Premier programme

```
%!PS-Adobe-2.0 EPSF-1.2
%%BoundingBox: 0 0 612 792


% Premier programme

/Times-Roman findfont
75 scalefont setfont

100 120 moveto
0.5 setgray
45 rotate
(Premier programme) show

showpage
```

Nous reparlerons plus tard des deux premières lignes de ce programme dont le résultat apparaît à la page suivante.

La troisième ligne du programme, qui commence par le caractère « % » est un commentaire ignoré par l'interpréteur.

La quatrième ligne fait appel à une police de caractères ; « findfont » demande à l'interpréteur de rechercher la police **Times** dans la ROM de l'imprimante.

La cinquième ligne précise la force de corps – en points – de la police, grâce à l'instruction « scalefont ». La police est alors installée en RAM grâce à la commande « setfont ».

La sixième ligne effectue un déplacement du point courant (*current point*) vers le point de coordonnées $(100, 120)$. Rappelons que l'origine est le coin inférieur gauche de la feuille et que l'unité est le point ou encore un septante-deuxième de pouce.

La ligne sept définit un niveau de gris pour l'impression des caractères. Ces niveaux vont de 0 (noir) à 1 (blanc).

À la ligne huit, l'opérateur « rotate » correspond à une rotation du nombre de degrés indiqués, à savoir 45, de « la ligne d'impression. »

La neuvième ligne produit l'impression, dans la mémoire image, des mots « Premier programme ». Le texte à imprimer doit se trouver entre parenthèses.

L'instruction « showpage » de la dernière ligne sert à effectuer le transfert vers la page papier.

4 Généralités sur `PostScript`

4.1 Introduction

`PostScript` possède de nombreux opérateurs qui contrôlent le traitement sur page papier d'un environnement graphique qui peut prendre trois aspects principaux :

- des objets textes de types, positions et orientations variées ;
- des figures géométriques qui peuvent définir des remplissages ;
- des dessins ou images digitalisés.

Ces informations graphiques peuvent subir des transformations mathématiques telles que rotations, translations et changements d'échelle. L'environnement graphique de la page est caractérisé par un ensemble d'objets qui subissent l'action d'opérateurs ou de commandes.

Un **opérateur** est une action qui attend un certain nombre d'arguments en vue de son exécution. Une **commande**, elle, ne nécessite pas d'argument.

`PostScript` utilise la **notation postfixée** (RPN : Reverse Polish Notation) avec quatre piles de travail et gère les structures classiques de programmation (contrôles, répétitions, boucles).

4.2 Les objets

4.2.1 Introduction

Un programme `PostScript` est constitué de caractères ASCII qui représentent les objets ou procédures décrivant la mémoire image.

Les caractères « espace », « tabulation » et « retour de chariot » ont la même vocation : celle de séparer les objets ou procédures. Ainsi,

```
/Times-Roman findfont
75 scalefont setfont
```

est équivalent à

```
/Times-Roman
        findfont
75
scalefont
            setfont
```

4.2.2 Caractères spéciaux

Certains caractères ont un rôle spécial dans le langage. Il s'agit des suivants :

% : il introduit un commentaire ignoré par l'interpréteur.
`% Premier programme`

( ) : les parenthèses délimitent une chaîne de caractères à imprimer.
`(Premier programme) show`
produit l'impresssion du texte « Premier programme ».

[ ] : les crochets délimitent un tableau.
`[1 0 3]` représente en fait un vecteur ligne à 3 dimensions (voir ci-dessous).

<> : ces deux séparateurs encadrent une chaîne de caractères donnée par son code ASCII hexadécimal.
`<50 72 65 6D 69 65 72 20 70 72 6F 67 72 61 6D 6D 65> show`
produit l'impression du texte « Premier programme ».

/ : introduit un nom littéral pour une variable ou une procédure (voir ci-après).
`/x 10 def`
affecte la valeur « 10 » à la variable « x ».

{ } : les accolades délimitent une procédure.
`/double {2 mul} def`
Cette syntaxe définit une procédure dont le nom est « double » et qui, lors de son appel, aura pour effet de doubler la valeur d'un nombre (voir à la page 33).

4.2.3 Nombres (*numbers*)

`PostScript` traite des valeurs numériques entières et réelles. Il permet aussi de travailler avec des nombres exprimés dans une base en utilisant la syntaxe « base#valeur ».

12 −71 4624 0 +13 sont des entiers.
−.007 31.3 27.0 212.3e5 3E−4 sont des réels.
8#12 16#AF2 2#1011 valent respectivement 10 (en base 8), 2 802 (en base 16) et 11 (en base 2).

4.2.4 Chaînes de caractères (*strings*)

Une chaîne est délimitée par des parenthèses entre lesquelles on peut utiliser n'importe quel caractère sauf « (», «) » et « \ » qui sont des caractères spéciaux.

```
(Voici un exemple)
(un autre : }[%)
```

Dans une chaîne, le caractère « \ » est utilisé comme **échappement** pour inclure des parenthèses ou des caractères spéciaux. Ainsi, à l'intérieur d'une chaîne,

`\(` produit le caractère « (».

`\)` produit le caractère «) ».

`\\` produit le caractère « \ ».

`\ddd` produit le caractère dont le code ASCII en octal est « ddd ».

`\` permet d'écrire sur plusieurs lignes sans provoquer de coupure dans ce qui est produit.

```
(Voici un exemple)
```

est équivalent à

```
(Voici \
un \
exemple)
```

Nous avons également vu qu'une chaîne pouvait être définie par les codes ASCII hexadécimaux de ses caractères. Dans ce dernier cas, les codes doivent être encadrés non plus par des parenthèses, mais par « < » et « > ».

4.2.5 Noms (*names*)

Toute occurence de caractères réguliers qui ne peut être interprétée comme un nombre est vue comme un objet « nom » ou plus précisément comme un nom « exécutable ». Tous les caractères excepté les délimiteurs et les blancs peuvent apparaître dans les noms.
Par exemple,

```
abc  Offset  23A  13-456  a.b  $MonDico  $$  @pattern
```

sont des noms valables.
Il faut prendre garde, lorsqu'on choisit un nom qui commence par des chiffres, à ce qu'il ne représente pas un nombre. Ainsi, 23A est un nom valable tandis que 23E1 est un nombre réel et 23#1, un nombre en base 23.
Un « / » (slash) introduit un nom « littéral ». Le slash ne fait pas partie du nom lui-même mais est un préfixe qui indique que ce qui suit est un objet nom littéral. Il ne peut pas y avoir de blanc entre / et le nom.

4.2.6 Tableaux (*arrays*)

Un tableau est une collection unidimensionnelle d'objets auxquels on accède au moyen d'un indice numérique. Contrairement à ce que l'on rencontre dans la plupart des autres langages, les tableaux `PostScript` peuvent être hétérogènes, c'est-à-dire qu'ils peuvent contenir des nombres, en même temps que des chaînes, des dictionnaires, ... Les éléments d'un tableau sont encadrés par les caractères « [» et «] ». Ainsi, `[123 /abc (xyz)]` est un tableau contenant l'objet entier 123, l'objet nom littéral abc et l'objet chaîne xyz.
Tous les tableaux sont indicés à partir de 0 ; ainsi, tout tableau de n éléments est indicé de 0 à $n-1$.
`PostScript` traite seulement des tableaux unidimensionnels mais il nous est loisible de construire des tableaux de dimensions supérieures en utilisant des tableaux comme éléments de tableaux.

4.2.7 Procédures

Elles sont délimitées par les caractères spéciaux « { » et « } ». Ce sont en fait des tableaux (*arrays*) exécutables. Nous en avons déjà donné un exemple plus haut.

4.2.8 Objets booléens

Leur valeur est *true* ou *false* ; ils sont les résultats des opérateurs relationnels et logiques.

4.2.9 Dictionnaires

Un « dictionnaire » est une table qui associe une clé à une valeur. Ses entrées sont des paires d'objets `PostScript` (clé, valeur).
Au niveau 1 du langage, la pile des dictionnaires ou *dictionary stack* contient à sa base deux dictionnaires particuliers qui sont « systemdict » et « userdict ».
Le « systemdict » ne peut pas être modifié et contient la définition des opérateurs standards du langage. Le « userdict » qui se situe au-dessus du précédent dans la pile, est un dictionnaire où l'on peut écrire (au moyen de l'opérateur « def ») les définitions des variables et procédures.
Il est possible de créer d'autres dictionnaires dans la pile des dictionnaires.

4.2.10 Opérateurs

Un objet « opérateur » est une action du langage `PostScript`. Les opérateurs ont des noms et sont définis dans le dictionnaire appelé « systemdict ». Par exemple, pour l'opérateur d'addition « add », dans le systemdict, à la clé « add » est associée la description de l'opération d'addition de deux nombres.

4.2.11 Autres types d'objets

Il existe encore bien d'autres objets tels que les tableaux compactés, les fichiers, les marques, l'objet *null*, les objets *save*, les objets *fontID* dont il ne sera pas question dans un premier temps.

4.2.12 Objets simples et composés

Les objets simples sont les booléens, les fontID, les entiers, les marques, les noms, l'objet *null*, les opérateurs, les réels.
Les objets composés sont les tableaux, les dictionnaires, les fichiers, les *save* et les chaînes.
Une différence essentielle entre les objets simples et les objets composés est le comportement de ceux-ci lors des opérations de copie d'objets. Lorsqu'un objet simple est copié, toutes ses parties le sont (type, attributs et valeur). Lors de la copie d'un objet composé, la valeur n'est pas copiée, mais partagée entre l'original et la copie. Cela implique que si l'on modifie la valeur de l'original, la valeur de la copie est aussi modifiée (cf. utilisation d'un pointeur en langage Pascal ou C).

4.2.13 Attributs des objets

En plus du type et de la valeur, un objet `PostScript` possède des attributs qui affectent son comportement lorsqu'il est exécuté ou lorsqu'on effectue des opérations sur lui.
Ainsi, un objet peut être « littéral » ou « exécutable ». Dans le premier cas, l'interpréteur le traite strictement comme une donnée et le place sur la pile des opérandes afin de l'utiliser comme opérande d'un opérateur qui suit. Dans le second cas, l'interpréteur l'exécute.
Les constantes entières, réelles et les chaînes sont toujours des objets littéraux. Les noms sont littéraux s'ils sont précédés de « / » et exécutables sinon.
...
Un autre attribut d'un objet est son « accessibilité (*access*) ». Celle-ci peut être illimitée, en lecture seule, en exécution seule, ...

4.3 Piles

Comme nous l'avons déjà dit, `PostScript` utilise la notation postfixée (RPN) et la notion de pile. Cette technique possède une syntaxe élémentaire qui facilite les mécanismes d'interprétation (qui sont alors linéaires). Une **pile** est une structure LIFO (*Last In First Out*). Le dernier élément qui a été mis au sommet de la pile est le premier qui sera utilisé par un opérateur (voir la section concernant les opérateurs).

L'interpréteur `PostScript`, au niveau 1 du langage, utilise quatre piles distinctes qui sont

1. - **la pile d'opérandes** (***operand stack***)
2. - **la pile des dictionnaires** (***dictionary stack***)
3. - **la pile d'exécution** (***execution stack***)
4. - **la pile des états graphiques** (***graphic state stack***)

4.3.1 La pile d'opérandes

Elle contient les opérandes et les résultats obtenus sur eux par les opérateurs. Lors de l'exécution d'un programme, l'interpréteur y empile (*push*) les objets qu'il reconnaît comme données littérales. Lorsqu'un opérateur requiert un ou plusieurs opérandes, l'interpréteur les obtient en les dépilant (*pop*) du sommet de la pile des opérandes. Le ou les résultats calculés par les opérateurs sont eux empilés (*push*) sur la pile d'opérandes.

4.3.2 La pile des dictionnaires

Elle ne contient que les objets dictionnaires. En particulier, on y trouve toutes les variables et les procédures définies par le programme sous forme d'un nom. Elle renferme aussi les définitions de tracé des caractères. Nous avons déjà dit qu'à sa base, il existe deux dictionnaires particuliers qui ne peuvent être dépilés, le « systemdict » et le « userdict », et que l'on pouvait en empiler d'autres.

4.3.3 La pile d'exécution

Elle contient tous les objets (principalement des procédures et des fichiers) en cours d'exécution. Un programme peut en fait enchaîner plusieurs objets. Lorsque l'interpréteur suspend l'exécution d'un objet pour en entreprendre un autre, il empile ce nouvel objet sur la pile d'exécution. Lorsqu'il en a terminé avec ce dernier, il le dépile et retrouve alors sur la pile d'exécution l'ancien objet au stade d'exécution où il se trouvait.

4.3.4 La pile des états graphiques

`PostScript` peut sauvegarder le contexte graphique dans lequel on se trouve (définition d'une police de caractères, force de corps, style, unité de mesure, épaisseur de trait, matrice de transformation, type de pointillé, ...). Cette sauvegarde s'empile dans la pile des états graphiques au moyen de la commande `gsave`. Après un traitement graphique dans un contexte différent, on peut récupérer l'ancien contexte en le dépilant grâce à la commande `grestore`. Il est possible d'empiler jusqu'à 31 états graphiques.

4.4 Opérateurs et commandes

Rappelons que nous avons déjà fait la distinction entre « opérateur » qui est une action attendant un certain nombre d'arguments en vue de son exécution et « commande », qui est également une action mais qui ne nécessite pas d'argument.

4.4.1 Opérateurs et commandes sur la pile d'opérandes

Il existe différents opérateurs et commandes qui réarrangent ou manipulent les objets sur la pile d'opérandes. cela peut être utile lorsque les résultats fournis par certains opérateurs doivent être utilisés comme arguments par d'autres opérateurs qui requièrent les opérandes dans un autre ordre.

`clear`

supprime tous les éléments de la pile.

`dup`

recopie au sommet de la pile le dernier élément de cette pile.

États successifs de la pile	
3	3
4	3 4
5	3 4 5
dup	3 4 5 5

`exch`

échange les deux éléments du sommet de la pile.

États successifs de la pile	
3	3
4	3 4
5	3 4 5
exch	3 5 4

`pop`

enlève l'élément au sommet de la pile.

États successifs de la pile	
3	3
4	3 4
5	3 4 5
pop	3 4

`count`

compte le nombre d'éléments de la pile et empile ce résultat.

États successifs de la pile	
3	3
4	3 4
5	3 4 5
count	3 4 5 3

`copy`

duplique une partie de la pile. C'est un opérateur qui nécessite un argument en haut de la pile : le nombre d'éléments à dupliquer.

	États successifs de la pile
3	3
4	3 | 4
5	3 | 4 | 5
2	3 | 4 | 5 | 2
copy	3 | 4 | 5 | 4 | 5

`roll`

effectue une permutation circulaire d'éléments de la pile. Cet opérateur nécessite deux arguments : le nombre d'éléments à permuter et le sens de la permutation (1 ou -1).

	États successifs de la pile
3	3
4	3 | 4
5	3 | 4 | 5
6	3 | 4 | 5 | 6
3	3 | 4 | 5 | 6 | 3
1	3 | 4 | 5 | 6 | 3 | 1
roll	3 | 6 | 4 | 5

	États successifs de la pile
3	3
4	3 | 4
5	3 | 4 | 5
6	3 | 4 | 5 | 6
3	3 | 4 | 5 | 6 | 3
-1	3 | 4 | 5 | 6 | 3 | -1
roll	3 | 5 | 6 | 4

`index`

permet de recopier au sommet de la pile un élément qui occupe une position quelconque dans la pile d'opérandes. Cet opérateur nécessite un argument : l'indice (dans la pile) de l'élément à dupliquer. L'élément situé au haut de la pile a l'indice 0.

	États successifs de la pile
3	3
4	3 4
5	3 4 5
6	3 4 5 6
2	3 4 5 6 2
index	3 4 5 6 4

`mark`

Cette commande place une marque sur la pile et s'utilise avec les commandes `counttomark` qui compte les éléments de la pile situés au-dessus de la marque et `cleartomark` qui enlève les éléments de la pile situés au-dessus de la marque, ainsi que la marque elle-même.

	États successifs de la pile
3	3
mark	3 mark
4	3 mark 4
5	3 mark 4 5
counttomark	3 mark 4 5 2
cleartomark	3

4.4.2 Opérateurs arithmétiques et mathématiques

L'opérateur dépile les arguments opérandes qui lui sont nécessaires, effectue les instructions associées (qui sont consignées, rappelons-le, dans le systemdict) et empile le résultat calculé au sommet de la pile.

Opérateurs arithmétiques nécessitant un argument

`abs`

renvoie la valeur absolue de l'argument.

	États successifs de la pile
3	3
−4	3 −4
abs	3 4
abs	3 4

`neg`

renvoie l'opposé de l'argument.

États successifs de la pile	
3	3
−4	3 | −4
neg	3 | 4
neg	3 | −4

`ceiling`

ou fonction plafond. Elle renvoie le plus petit entier supérieur ou égal à l'argument.

États successifs de la pile	
3.4	3.4
ceiling	4
−4.7	4 | −4.7
ceiling	4 | −4

`floor`

ou fonction plancher. Elle renvoie le plus grand entier inférieur ou égal à l'argument.

États successifs de la pile	
4.3	4.3
floor	4
−3.8	4 | −3.8
floor	4 | −4

`round`

ou fonction arrondi. Elle renvoie la valeur arrondie de l'argument.

	États successifs de la pile
3.2	3.2
round	3
6.5	3 | 6.5
round	3 | 7
−4.7	3 | 7 | −4.7
round	3 | 7 | −5
−6.5	3 | 7 | −5 | −6.5
round	3 | 7 | −5 | −6

`truncate`

ou fonction troncature. Elle renvoie la valeur tronquée de l'argument obtenue par suppression de la partie fractionnaire.

	États successifs de la pile
4.7	4.7
truncate	4
−3.9	4 | −3.9
truncate	4 | −3

Opérateurs arithmétiques nécessitant deux arguments

`add`

effectue la somme des deux arguments.

	États successifs de la pile
3	3
4	3 | 4
add	7

`sub`

effectue la différence des deux arguments.

	États successifs de la pile
3	3
4	3 | 4
sub	−1

`mul`

effectue le produit des deux arguments.

	États successifs de la pile
3	3
4	3 | 4
mul	12

`div`

effectue le quotient des deux arguments.

	États successifs de la pile
3	3
4	3 | 4
div	0.75

`idiv`

effectue la division entière des deux arguments.

	États successifs de la pile
5	5
3	5 | 3
idiv	1

`mod`

donne le reste de la division entière des deux arguments.

	États successifs de la pile
5	5
3	5 | 3
mod	2

Les opérateurs de recherche du maximum et du minimum de deux nombres n'existent pas, mais peuvent être créés par l'utilisateur, comme nous le ferons à la page 33.

Opérateurs mathématiques et fonctions trigonométriques

`sqrt`

renvoie la racine carrée d'un nombre qui doit obligatoirement être positif.

États successifs de la pile	
5	5
2	5 | 2
sqrt	5 | 1.41421

`exp`

nécessite deux arguments : une *base* et un *exposant*. Cet opérateur élève la base à la puissance de l'exposant ; si l'exposant possède une partie fractionnaire, le résultat n'a de sens que si la base est non négative.

États successifs de la pile	
9	9
−0.5	9 | −0.5
exp	0.333333

`ln`

renvoie le logarithme naturel ou népérien de l'argument.

États successifs de la pile	
5	5
2	5 | 2
ln	5 | 0.69315

`log`

renvoie le logarithme décimal de l'argument.

États successifs de la pile	
5	5
2	5 | 2
log	5 | 0.30103

`sin`

renvoie le sinus de l'argument exprimé en degrés.

	États successifs de la pile
5	5
20.49	5 | 20.49
sin	5 | 0.350044

`cos`

renvoie le cosinus de l'argument exprimé en degrés.

	États successifs de la pile
5	5
20.49	5 | 20.49
cos	5 | 0.936733

`atan`

nécessite deux arguments ; appelons-les *num* et *den*. Cet opérateur renvoie l'angle exprimé en degrés et compris entre 0 et 360 dont la tangente est égale à *num* divisé par *den*. Les arguments *num* ou *den* peuvent être nuls mais pas en même temps. Les signes de *num* et de *den* déterminent le quadrant dans lequel se trouve le résultat : un *num* positif donne un résultat dans le demi-plan des y positifs et un *den* positif, un résultat dans le demi-plan des x positifs.

	États successifs de la pile
5	5
0	5 | 0
1	5 | 0 | 1
atan	5 | 0.0
1	5 | 0.0 | 1
0	5 | 0.0 | 1 | 0
atan	5 | 0.0 | 90.0
−100	5 | 0.0 | 90.0 | −100
0	5 | 0.0 | 90.0 | −100 | 0
atan	5 | 0.0 | 90.0 | 270.0
4	5 | 0.0 | 90.0 | 270.0 | 4
4	5 | 0.0 | 90.0 | 270.0 | 4 | 4
atan	5 | 0.0 | 90.0 | 270.0 | 45.0

Nombres aléatoires

`PostScript` possède des commandes et opérateurs qui permettent de manipuler des nombres aléatoires. Il s'agit de

`rand` qui renvoie un nombre entier aléatoire compris entre 0 et $2^{31} - 1$;

`rrand` qui renvoie un entier représentant l'état actuel du générateur de nombres aléatoires utilisé par `rand` ;

`srand` qui est un opérateur nécessitant un argument à partir duquel il initialise le générateur de nombres aléatoires.

4.4.3 Autres opérateurs

Certains opérateurs sont *polymorphes* : ils s'appliquent à différents types d'opérandes mais leur action peut différer selon le type. Nous en donnons une description ci-dessous.

Opérateurs sur les tableaux

`length`

renvoie le nombre d'éléments du tableau unidimensionnel.

États successifs de la pile	
5	5
[7 3 (xy) 1]	5 | [7 3 (xy) 1]
length	5 | 4

`get`

utilise deux arguments : un tableau et un indice. L'opérateur empile l'élément du tableau dont l'indice est précisé (rappelons que la numérotation des indices démarre à 0).

États successifs de la pile	
5	5
[7 3 9 1]	5 | [7 3 9 1]
2	5 | [7 3 9 1] | 2
get	5 | 9

`array`

crée un tableau vide. Il nécessite un argument : la longueur du tableau.

États successifs de la pile	
5	5
3	5 | 3
array	5 | [. . .]

`astore`

place des éléments de la pile dans un tableau.

	États successifs de la pile
5	5
7	5 | 7
3	5 | 7 | 3
9	5 | 7 | 3 | 9
1	5 | 7 | 3 | 9 | 1
4	5 | 7 | 3 | 9 | 1 | 4
array	5 | 7 | 3 | 9 | 1 | [. . . .]
astore	5 | [7 3 9 1]

La syntaxe suivante est aussi valable : `/tab 4 array def`. Elle définit une variable dont le nom est `tab` qui est un tableau vide à quatre éléments. Celui-ci pourra être rempli au moyen de `astore`.

	États successifs de la pile
5	5
7	5 | 7
3	5 | 7 | 3
9	5 | 7 | 3 | 9
1	5 | 7 | 3 | 9 | 1
tab	5 | 7 | 3 | 9 | 1 | [. . . .]
astore	5 | [7 3 9 1]

`aload`

place les éléments d'un tableau dans la pile et recopie le tableau au sommet de la pile.

	États successifs de la pile
5	5
[7 3 9 1]	5 | [7 3 9 1]
aload	5 | 7 | 3 | 9 | 1 | [7 3 9 1]

`getinterval`

nécessite trois arguments : un tableau, un indice et le nombre d'éléments à extraire du tableau à partir de l'indice de manière à former un nouveau tableau que l'opérateur place au sommet de la pile.

	États successifs de la pile
5	5
[7 3 9 1]	5 | [7 3 9 1]
1	5 | [7 3 9 1] | 1
2	5 | [7 3 9 1] | 1 | 2
getinterval	5 | [3 9]

`copy`

nécessite deux arguments : deux tableaux. Le deuxième tableau doit être stocké dans une variable. L'opérateur substitue le premier tableau comme sous-tableau à partir du début du deuxième tableau.

`/tab 6 array def` définit une variable dont le nom est `tab`, tableau vide à 6 éléments qui pourra être rempli au moyen de l'opérateur `astore`.

	États successifs de la pile
5	5
[7 3 9 1]	5 | [7 3 9 1]
9	5 | [7 3 9 1] | 9
8	5 | [7 3 9 1] | 9 | 8
7	5 | [7 3 9 1] | 9 | 8 | 7
5	5 | [7 3 9 1] | 9 | 8 | 7 | 5
4	5 | [7 3 9 1] | 9 | 8 | 7 | 5 | 4
3	5 | [7 3 9 1] | 9 | 8 | 7 | 5 | 4 | 3
tab	5 | [7 3 9 1] | 9 | 8 | 7 | 5 | 4 | 3 | [. . . . . .]
astore	5 | [7 3 9 1] | [9 8 7 5 4 3]
copy	5 | [7 3 9 1]

La variable `tab` représente alors le tableau `[7 3 9 1 4 3]`.

`put`

nécessite trois arguments : un tableau (stocké dans une variable), un indice et quelque chose à placer dans le tableau à l'indice précisé. Il faut donc d'abord une instruction du type `/tab 4 array def`, qui définit la variable `tab` comme tableau de 4 éléments.

	États successifs de la pile
5	5
[7 3 9 1]	5 | [7 3 9 1]
tab	5 | [7 3 9 1] | [. . . .]
copy	5 | [7 3 9 1]
2	5 | [7 3 9 1] | 2
(xyz)	5 | [7 3 9 1] | 2 | (xyz)
put	5

La variable `tab` représente alors le tableau `[7 3 (xyz) 1]`.

`putinterval`

nécessite trois arguments : un tableau (stocké dans une variable), un indice et un sous-tableau à placer dans le tableau à l'indice précisé. Il faut donc d'abord une instruction du type `/tab 4 array def`, qui définit la variable `tab` comme tableau de 4 éléments. Le sous-tableau est alors substitué dans le premier tableau à partir de l'indice précisé.

	États successifs de la pile
5	5
[7 3 9 1]	5 | [7 3 9 1]
tab	5 | [7 3 9 1] | [. . . .]
copy	5 | [7 3 9 1]
1	5 | [7 3 9 1] | 1
[8 2]	5 | [7 3 9 1] | 1 | [8 2]
putinterval	5

La variable `tab` représente alors le tableau `[7 8 2 1]`.

Opérateurs sur les chaînes

De nombreux opérateurs dont l'action vient d'être décrite sur les tableaux ont un comportement relativement semblable sur les chaînes.

`length`

renvoie le nombre d'éléments de la chaîne.

	États successifs de la pile
5	5
`(abc28@mp}#yZ'\312\))`	5 | `(abc28@mp}#yZ'\312\))`
length	5 | 15

`get`

utilise deux arguments : une chaîne et un indice. L'opérateur empile le code ASCII décimal de l'élément de la chaîne dont l'indice est précisé (rappelons que la numérotation des indices démarre à 0).

États successifs de la pile	
5	[5]
(PostScript)	[5] [(PostScript)]
6	[5] [(PostScript)] [6]
get	[5] [114]

114 est le code décimal ASCII du caractère « r ».

`string`

crée une chaîne vide. Il nécessite un argument : la longueur de la chaîne.

États successifs de la pile	
5	[5]
3	[5] [4]
string	[5] [(....)]

`getinterval`

nécessite trois arguments : une chaîne, un indice et le nombre d'éléments à extraire de la chaîne à partir de l'indice pour former une nouvelle chaîne que l'opérateur place au sommet de la pile.

États successifs de la pile	
5	[5]
(ABCDEFG)	[5] [(ABCDEFG)]
2	[5] [(ABCDEFG)] [2]
3	[5] [(ABCDEFG)] [2] [3]
getinterval	[5] [(CDE)]

`copy`

nécessite deux arguments : deux chaînes. La deuxième chaîne doit être stockée dans une variable. L'opérateur substitue la première chaîne comme sous-chaîne à partir du début de la deuxième chaîne.

`/alpha (ABCDEFG) def` définit une variable dont le nom est `alpha`, qui contient la chaîne `(ABCDEF)`.

	États successifs de la pile
5	5
/alpha	5 | /alpha
(ABCDEFG)	5 | /alpha | (ABCDEFG)
def	5
(xyz)	5 | (xyz)
alpha	5 | (xyz) | (ABCDEFG)
copy	5 | (xyz)

La variable `alpha` représente alors la chaîne `(xyzDEFG)`.

`put`

nécessite trois arguments : une chaîne (stockée dans une variable), un indice et le code ASCII (en décimal) d'un caractère à placer dans la chaîne à l'indice précisé.

	États successifs de la pile
5	5
/alpha	5 | /alpha
(ABCDEFG)	5 | /alpha | (ABCDEFG)
def	5
alpha	5 | (ABCDEFG)
4	5 | (ABCDEFG) | 4
101	5 | (ABCDEFG) | 4 | 101
put	5

La variable `alpha` représente alors la chaîne `(ABCDeFG)` (101 est le code ASCII décimal du caractère « e » (minuscule).

`putinterval`

nécessite trois arguments : une chaîne (stockée dans une variable), un indice et une sous-chaîne à substituer dans la chaîne à partir de l'indice précisé.

	États successifs de la pile
5	5
/alpha	5 | /alpha
(ABCDEFG)	5 | /alpha | (ABCDEFG)
def	5
alpha	5 | (ABCDEFG)
2	5 | (ABCDEFG) | 2
(xyz)	5 | (ABCDEFG) | 2 | (xyz)
putinterval	5

La variable `alpha` représente alors la chaîne `(ABxyzFG)`.

`search`

nécessite deux arguments : deux chaînes, la première étant celle dans laquelle on recherche l'occurrence de la deuxième. Si cette occurrence est trouvée, l'opérateur empile, la sous-chaîne qui suit l'occurence, l'occurence elle-même, la sous-chaîne qui précède l'occurence et la valeur booléenne `true`. Dans le cas contraire, il empile la première chaîne et la valeur booléenne `false`.

	États successifs de la pile
5	5
(ABCDEFG)	5 | (ABCDEFG)
(ABC)	5 | (ABCDEFG) | (ABC)
search	5 | (DEFG) | (ABC) | () | true
pop	5 | (DEFG) | (ABC) | ()
pop	5 | (DEFG) | (ABC)
pop	5 | (DEFG)
(FE)	5 | (DEFG) | (FE)
search	5 | (DEFG) | false
pop	5 | (DEFG)
(EF)	5 | (DEFG) | (EF)
search	5 | (G) | (EF) | (D) | true

`anchorsearch`

nécessite deux arguments : deux chaînes. On examine si la première commence par la deuxième. Si c'est le cas, l'opérateur empile, la sous-chaîne qui suit l'occurence, l'occurence

elle-même et la valeur booléenne `true`. Dans le cas contraire, il empile la première chaîne et la valeur booléenne `false`.

États successifs de la pile	
5	5
(ABCDEFG)	5 | (ABCDEFG)
(ABC)	5 | (ABCDEFG) | (ABC)
anchorsearch	5 | (DEFG) | (ABC) | true
pop	5 | (DEFG) | (ABC)
pop	5 | (DEFG)
(EF)	5 | (DEFG) | (EF)
anchorsearch	5 | (DEFG) | false
pop	5 | (DEFG)
(d)	5 | (DEFG) | (d)
anchorsearch	5 | (DEFG) | false

4.4.4 Opérateurs booléens

`and`

renvoie la conjonction logique des deux arguments qu'il nécessite. Cet opérateur agit sur des valeurs booléennes ou entières.

États successifs de la pile	
5	5
true	5 | true
true	5 | true | true
and	5 | true
483	5 | true | 483
287	5 | true | 483 | 287
and	5 | true | 259

En fait,

483 s'écrit 111100011 en binaire
287 s'écrit 100011111 en binaire

et la conjonction logique est 259 puisque

259 s'écrit 100000011 en binaire.

`or`

renvoie la disjonction logique des deux arguments qu'il nécessite. Cet opérateur agit sur des valeurs booléennes ou entières.

	États successifs de la pile
5	5
false	5 | false
true	5 | false | true
or	5 | true
481	5 | true | 481
285	5 | true | 481 | 285
or	5 | true | 509

En fait,

481 s'écrit 111100001 en binaire
285 s'écrit 100011101 en binaire

et la disjonction logique est 509 puisque

509 s'écrit 111111101 en binaire.

`xor`

renvoie la disjonction exclusive logique des deux arguments qu'il nécessite. Cet opérateur agit sur des valeurs booléennes ou entières.

	États successifs de la pile
5	5
true	5 | true
true	5 | true | true
xor	5 | false
481	5 | false | 481
285	5 | false | 481 | 285
xor	5 | false | 252

En fait,

481 s'écrit 111100001 en binaire
285 s'écrit 100011101 en binaire

et la disjonction exclusive logique est 252 puisque

252 s'écrit 011111100 en binaire.

`not`

renvoie la négation logique de l'argument qu'il nécessite. Cet opérateur agit sur des valeurs booléennes ou entières. Dans ce dernier cas, il empile le complément à 1.

	États successifs de la pile
5	5
true	5 | true
not	5 | false
−124	5 | false | −124
not	5 | false | 123

En fait,

−124 s'écrit
11111111 11111111 11111111 10000100
en binaire (complément à 2)

et le complément à 1 de cette expression binaire est
00000000 00000000 00000000 01111011

qui représente effectivement 123.

`eq`

teste l'égalité des deux arguments et empile une valeur booléenne (`true` ou `false`).

	États successifs de la pile
5	5
7	5 | 7
7.0	5 | 7 | 7.0
eq	5 | true

Il est aussi possible d'utiliser les opérateurs `ne` (non-égalité), `ge` (plus grand ou égal à), `gt` (strictement plus grand que), `le` (plus petit ou égal à) et `lt` (strictement plus petit que).

	États successifs de la pile
5	5
(uvw)	5 | (uvw)
(uv)	5 | (uvw) | (uv)
ge	5 | true

	États successifs de la pile
5	5
(uvw)	5 | (uvw)
(z)	5 | (uvw) | (z)
ge	5 | false

`true` et `false` empilent les valeurs booléennes que ces constantes représentent.

Il est également possible d'opérer une permutation des bits au moyen de l'opérateur `bitshift` qui nécessite deux arguments (un entier) et un pas entier (un pas positif opère un « shift » vers la gauche).

4.4.5 Opérateurs de contrôle

`if` et `ifelse`

permettent l'exécution d'une procédure `procedure` compte tenu du résultat d'une expression de type booléen `condition`.

```
condition {procedure} if
```

Cet opérateur exécute la procédure si la condition est vraie.

```
condition {procedure1} {procedure2} ifelse
```

Cet opérateur exécute la première procédure si la condition est vraie et la seconde si la condition est fausse.

`for`

Il s'agit d'une structure de boucle qui répète une procédure depuis une valeur initiale (`vinit`) jusqu'à une valeur finale (`vfin`) avec une incrémentation (`incr`).

```
vinit incr vfin {procedure} for
```

Il faut savoir que la variable de contrôle qu'on appelle encore pointeur d'exécution est placée, pour chaque boucle, au sommet de la pile d'opérandes. Ceci est bien utile dans la cas où la procédure « consomme » la variable de contrôle.

`exit`

permet de sortir d'une boucle de manière prématurée, par exemple, au moyen d'un test.

```
vinit incr vfin { procedure condition {exit} if } for
```

termine l'exécution de la structure répétitive et libère la pile de toute référence à cette dernière lorsque la condition est vraie.

`repeat`

répète une procédure un certain nombre de fois qui est indiqué au moyen d'un entier `integer`.

`integer {procedure} repeat`

Il est utile de savoir que cet opérateur n'affecte pas la pile durant son exécution.

`loop`

est une boucle répétitive qui n'est pas contrôlée par une variable. De cette structure qui n'affecte pas la pile durant son exécution, on ne peut sortir que grâce à la commande `exit`. Cet opérateur combiné avec l'emploi de la commande `exit` permet la mise en place de deux structures répétitives :

« Répéter jusqu'à » `condition`

```
{
procedure
condition {exit} if
} loop
```

« Tant que » `non-condition` « faire »

```
{
condition {exit} if
procedure
} loop
```

`forall`

nécessite deux opérandes (une chaîne ou un tableau) et une procédure. L'opérateur énumère les éléments de la chaîne ou du tableau et leur fait subir à chacun la procédure.

Nous donnons ci-dessous deux exemples d'utilisation de l'opérateur `forall`. Dans le premier cas, l'action a lieu sur une chaîne de caractères (décomposition de celle-ci) et dans le second, sur un tableau (opération de multiplication des composantes d'un vecteur par un scalaire).

```
%!PS-Adobe-2.0 EPSF-1.2
%%BoundingBox: 0 0 590 780

100 500 moveto

/Times-Roman findfont
35 scalefont setfont

/s 1 string def
/alphabet (ABCDEFGHIJKL) def

alphabet
{
/code exch def
s 0 code put
s show
(-) show
} forall

showpage
```

Ce programme décompose la chaîne `/alphabet` caractère par caractère et imprime ceux-ci en intercalant le caractère « - » entre chaque caractère de la chaîne.

```
/mulsc
{
exch
/@sc exch def
dup
length
/@lvct exch def
{@sc mul} forall
@lvct array astore
} def
```

Cette procédure nécessite deux arguments (un réel et un tableau). Elle renvoie un tableau dans lequel chaque composante du tableau donné a été multipliée par le réel donné.

4.4.6 Opérateurs de conversion

`type`

nécessite un argument et renvoie un objet « nom » qui définit le type de l'argument. Celui-ci peut être `arraytype`, `booleantype`, `integertype`, `marktype`, `nametype`, `operatortype`, `realtype`, `stringtype`, ...

`cvi`

nécessite un argument (nombre ou chaîne numérique) qu'il convertit en entier.

`cvr`

nécessite un argument (nombre ou chaîne numérique) qu'il convertit en réel.

`cvs` et `cvrs`

Ces deux opérateurs nécessitent, avant d'être utilisés, que l'on crée une chaîne de caractères d'une certaine longueur au moyen de la syntaxe

```
/chn 50 string def
```

* `cvs` convertit alors son argument en chaîne et le stocke dans la chaîne `/chn` qui a été définie :

  ```
  543 702 add chn cvs
  ```

  produit l'attribution de la valeur (1245) à la variable `chn`.
* `cvrs` convertit alors un entier dans une base entière et le stocke dans la chaîne `/chn` qui a été définie :

  ```
  203 16 chn cvrs
  ```
  produit l'attribution de la valeur (CB) à la variable `chn`

  ```
  203 2 chn cvrs
  ```
  produit l'attribution de la valeur (11001011) à la variable `chn`.

Il faut savoir que outre les dessins, `PostScript` ne peut imprimer que des chaînes de caractères. Il faut d'abord indiquer où imprimer la chaîne sur la page au moyen de l'opérateur `moveto` (voir page 37), ensuite empiler la chaîne et la commande `show`.

5 Variables et procédures

5.1 Variables

Il est très simple d'affecter une valeur à une variable au moyen de la commande `def`, éventuellement combinée avec la commande `exch`.

	États successifs de la pile
5	5
/var	5 | /var
9	5 | /var | 9
def	5
var	5 | 9

À ce moment, dans le `userdict`, la variable `var` est stockée avec la valeur 9. Elle peut donc être réutilisée tout au long du programme.
Il arrive souvent qu'une valeur que l'on désire attribuer à une variable se trouve en sommet de pile. On utilise alors la succession des commandes `exch` et `def`, de manière à placer, dans la pile

d'opérandes, le nom littéral de la variable avant la valeur qu'on veut lui attribuer et se retrouver ainsi dans la situation précédente.

	États successifs de la pile
5	5
9	5 | 9
/var	5 | 9 | /var
exch	5 | /var | 9
def	5
var	5 | 9

Pour les variables de type tableau, on peut aussi utiliser l'opérateur `astore` (voir à la page 20) ou l'opérateur `copy` (voir à la page 22).

5.2 Procédures

Nous avons déjà rencontré à la page 6 la syntaxe qui permet de définir une procédure. Reprenons cet exemple et montrons les états successifs de la pile lors de l'utilisation de cette procédure.

```
/double {2 mul} def
```

Cette procédure dont la définition est stockée dans le `userdict` a pour effet de doubler la valeur du nombre qui se trouve au sommet de la pile des opérandes.

	États successifs de la pile
5	5
9	5 | 9
double	5 | 18

Certaines procédures, plus complexes, nécessitent l'emploi de plusieurs variables pour réaliser leur tâche. Il est alors intéressant de passer ces variables comme arguments à la procédure. La syntaxe est semblable à celle que nous avons rencontrée ci-dessus lors de l'attribution de valeurs aux variables. Nous l'illustrons par quelques exemples qui enrichissent en même temps notre bibliothèque de procédures mathématiques.
Nous avons déjà signalé à la page 16 que `PostScript` ne possède pas les opérateurs de recherche du maximum et du minimum de deux nombres. Voici une manière de s'y prendre.

```
/max {/@op2 exch def /@op1 exch def
@op1 @op2 gt {@op1} {@op2} ifelse} def

/min {/@ope2 exch def /@ope1 exch def
@ope1 @ope2 lt {@ope1} {@ope2} ifelse} def
```

Ces nouveaux opérateurs définis par l'utilisateur agissent de la même manière que les opérateurs prédéfinis de `PostScript` en ce sens qu'ils dépilent les arguments qui leur sont nécessaires et empilent le résultat. Ainsi, `7 5 min` dépile d'abord le 5, ensuite le 7 (qui sont placés dans des variables transmises comme arguments à la procédure) et empile 5.

Voici encore les opérateurs qui fournissent le plus grand commun diviseur et le plus petit commun multiple de deux nombres entiers.

```
/pgcd
{
/@oper2 exch def
/@oper1 exch def
{
@oper2 0 eq {exit} if
/@temp @oper1 @oper2 mod def
/@oper1 @oper2 def
/@oper2 @temp def
} loop
@oper1 abs
} def

/ppcm
{
/@opera2 exch def
/@opera1 exch def
@opera1 dup @opera2 dup 4 -1 roll
pgcd idiv mul abs
} def
```

Il est également possible de définir l'addition vectorielle composante par composante quelle que soit la dimension de l'espace vectoriel considéré, de même, la soustraction vectorielle, la multiplication par un scalaire et de créer un opérateur de produit scalaire (en base orthonormée).

```
/addn
{/@vct2 exch def
dup
/@vct1 exch def
length
/@lvct exch def
/@ind 0 def
@lvct {
@vct1 @ind get
@vct2 @ind get add
/@ind @ind 1 add def
} repeat
@lvct array astore
} def
```

```
/subn
{/@vct2 exch def
dup
/@vct1 exch def
length
/@lvct exch def
/@ind 0 def
@lvct {
@vct1 @ind get
@vct2 @ind get sub
/@ind @ind 1 add def
} repeat
@lvct array astore
} def
```

```
/mulsc
{exch
/@sc exch def
dup
length
/@lvct exch def
{@sc mul} forall
@lvct array astore
} def
```

```
/prdsc
{/@vct2 exch def
dup
/@vct1 exch def
length
/@lvct exch def
/@ind 0 def
0
@lvct {
@vct1 @ind get
@vct2 @ind get mul add
/@ind @ind 1 add def
} repeat
} def
```

6 Graphisme en PostScript

6.1 Transformations

L'environnement graphique de PostScript utilise un système de coordonnées (*user space*) qui est en relation directe avec celui de l'imprimante connectée (*device space*). Le point origine $(0, 0)$ est situé dans le coin inférieur gauche, les axes sont situés sur les bords de la feuille et l'unité de mesure est le « point » qui vaut $1/72^{\text{ème}}$ de pouce. Cet espace utilisateur peut être modifié par des **translations**, des **rotations** ou des **changements d'échelle**. Il est important de savoir que ces différents opérateurs n'agissent pas commutativement.
En fait, le contexte graphique tient à jour une matrice appelée CTM ou *current transformation matrix* qui envoie les coordonnées spécifiées par le programme PostScript vers le repère du périphérique. Elle agit de la manière suivante :

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

Les modifications successives du contexte graphique se traduisent ainsi par des produits matriciels.
Au début d'un programme, cette matrice CTM est par défaut la matrice identité

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Il existe de nombreux opérateurs qui agissent directement sur cette matrice mais, dans une première approche, nous ne parlerons que des trois décrits ci-dessous.

6.1.1 Translation

L'utilisateur peut déplacer l'origine du repère à l'aide de l'opérateur `translate` qui nécessite deux arguments, l'abscisse et l'ordonnée du point vers lequel on translate cette origine. La

syntaxe est

```
100 250 translate
```

Ceci a pour effet de déplacer l'origine du repère depuis $(0,0)$ jusqu'à $(100,250)$, l'unité étant le « point ». La matrice associée a la forme

$$\begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}.$$

6.1.2 Rotation

L'utilisateur peut également effectuer une rotation du système d'axes autour de son origine au moyen de l'opérateur `rotate` qui nécessite un argument : l'angle de la rotation exprimé en degrés. La syntaxe est

```
45 rotate
```

Ceci a pour effet de faire tourner les axes de 45 degrés dans le sens trigonométrique. Pour obtenir une rotation dans l'autre sens, il suffit de fournir à l'opérateur un argument négatif. La matrice associée a la forme

$$\begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

6.1.3 Changement d'échelle

L'utilisateur a aussi la possibilité de changer l'échelle sur chacun des deux axes au moyen de l'opérateur `scale` qui nécessite deux arguments, le facteur de changement d'échelle sur chacun des axes. La syntaxe est

```
20 30 scale
```

Ceci a pour effet de multiplier par 20 l'unité sur l'axe des abscisses et par 30 celle sur l'axe des ordonnées. La matrice associée à la forme

$$\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Pour travailler en centimètres, on définit le facteur `s` qui représente le nombre de points par centimètre et on l'utilise comme facteur d'échelle sur les deux axes, ce qui donne :

```
/s 72 2.54 div def
s dup scale
```

6.2 Chemin (*path*)

Un chemin `PostScript` ou *path* est constitué par un ensemble de lignes droites ou courbes qui déterminent une trajectoire. Celle-ci peut être continue, discontinue, ouverte, fermée, ...

Le contexte graphique de la page `PostScript` est caractérisé par un chemin courant (*current path*). Pour en définir un nouveau, il est nécessaire de réinitialiser le chemin courant à l'aide de la commande `newpath`.
Lors de l'initialisation du chemin courant, aucun point de départ n'est *a priori* défini. Ce point de départ, aussi appelé point courant (*current point*), est introduit au moyen de l'opérateur `moveto` qui nécessite deux arguments : l'abscisse et l'ordonnée du point courant.

`moveto` nécessite deux arguments qui sont l'abscisse et l'ordonnée d'un point de la page.
S'il suit un `newpath`, il définit un nouveau point courant qui démarre un nouveau chemin. À l'intérieur d'un chemin, il déplace le point courant, ce qui permet notamment de tracer un chemin discontinu.

`rmoveto` nécessite deux arguments qui sont les composantes d'un vecteur qui détermine le déplacement du point courant. Il ne peut s'utiliser que si un point courant existe déjà.

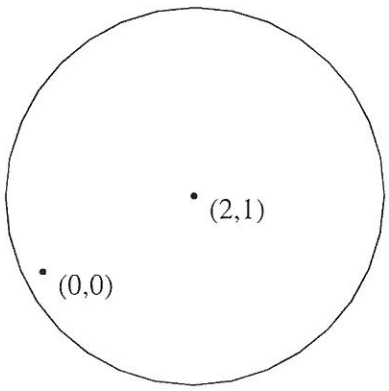
`lineto` nécessite deux arguments qui sont l'abscisse et l'ordonnée d'un point de la page. Cet opérateur définit un segment de droite qui va du point courant au point indiqué. Cette commande ne dessine pas le segment.

`rlineto` nécessite deux arguments qui sont les composantes d'un vecteur qui définit un segment de droite dont l'origine est au point courant. Le segment n'est pas dessiné.

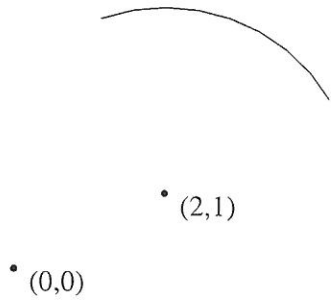
`closepath` définit le segment qui ferme le chemin courant depuis le point courant jusqu'au point initial du chemin courant.

`arc` définit un arc de cercle qu'il ajoute au chemin courant. Il nécessite cinq arguments. Les deux premiers sont les coordonnées du centre du cercle, le troisième est la longueur du rayon et les deux suivants sont les valeurs en degrés délimitant le début et la fin de l'arc. Cet opérateur agit dans le sens trigonométrique.
Dans les exemples ci-dessous, nous avons travaillé en centimètres.

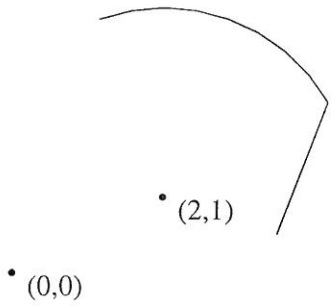
```
newpath
2 1 2.5 0 360 arc
stroke
```



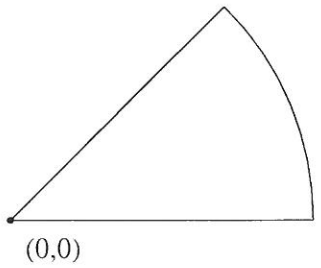
```
newpath
2 1 2.5 30 110 arc
stroke
```




```
newpath
3.5 0.5 moveto
2 1 2.5 30 110 arc
stroke
```

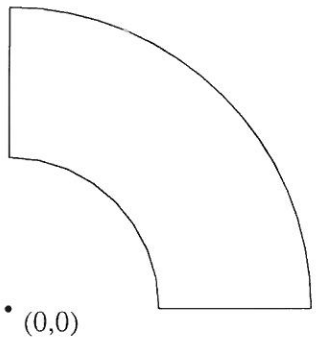



```
newpath
0 0 moveto
0 0 4 0 45 arc
closepath
stroke
```



L'opérateur `arcn` agit comme `arc` mais dans le sens opposé au sens trigonométrique.

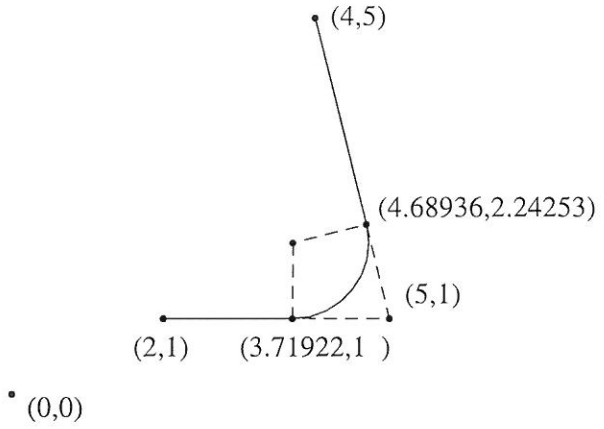
```
newpath
0 0 4 0 90 arc
0 0 2 90 0 arcn
closepath
stroke
```



`arct` nécessite cinq arguments, les coordonnées de deux points (dans notre exemple, $(5,1)$ et $(4,5)$) et un rayon (1 dans l'exemple). `arct` ajoute un arc de rayon 1 tangent intérieurement au contour allant du point courant (dans notre exemple, $(2,1)$) au point $(5,1)$ puis au point $(4,5)$. Une variante de cet opérateur est `arcto`, qui nécessite également cinq arguments, qui agit de la même façon mais empile les coordonnées des points de contact qui

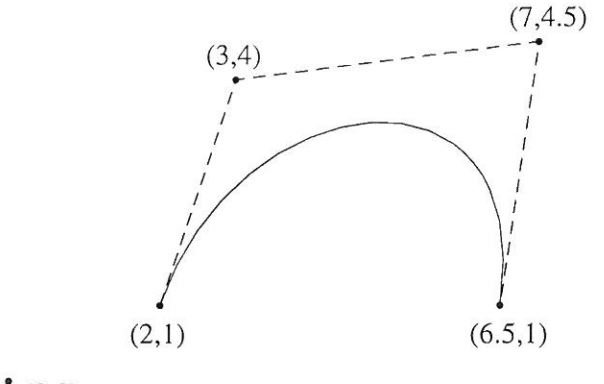
sont en réalité les extrémités de l'arc. Dans notre exemple, la pile contient alors de bas en haut 3.71922 | 1.0 | 4.68937 | 2.24254.

```
newpath
2 1 moveto
5 1 4 5 1 arct
4 5 lineto
stroke
```

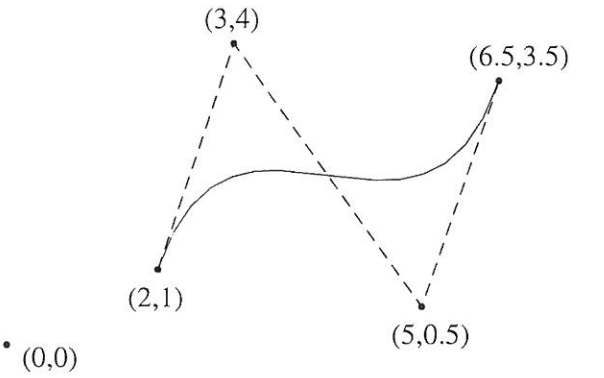


`curveto` nécessite six arguments : les coordonnées de trois points. L'opérateur ajoute, à partir du point courant une courbe de Bézier (cubique donnée sous forme paramétrique). Celle-ci est tangente au segment qui va du point courant au premier des trois points et au segment qui va du deuxième au troisième point. De plus, elle est entièrement contenue dans le quadrilatère défini par le point courant et les trois points donnés. En voici trois exemples.

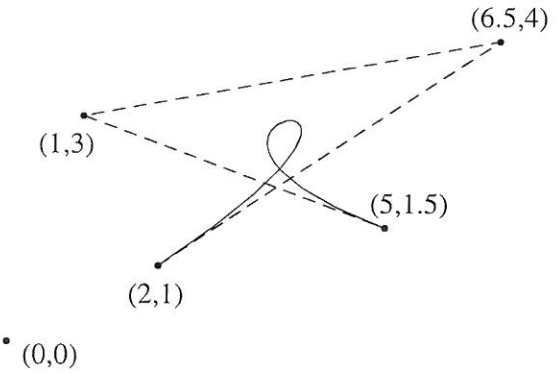
```
newpath
2 1 moveto
3 4 7 4.5 6.5 1 curveto
stroke
```




```
newpath
2 1 moveto
3 4 5 0.5 6.5 3.5 curveto
stroke
```



```
newpath
2 1 moveto
6.5 4 1 3 5 1.5 curveto
stroke
```



`stroke` est une commande qui dessine les segments ou les courbes définis préalablement dans le chemin courant.

`fill` est une commande qui peint l'aire à l'intérieur du chemin courant au moyen de la couleur courante. Elle utilise un algorithme pour déterminer ce qui est intérieur et ce qui ne l'est pas. Cette règle, appelée en anglais *Nonzero Winding Number Rule*, est assez subtile et, dans un premier temps, nous nous limiterons à ce que nous dicte le bon sens. Après avoir rempli l'intérieur du chemin courant, `fill` ne le maintient plus comme chemin courant car il réalise implicitement un `newpath`. Si on veut préserver le chemin courant lors d'une opération de remplissage, on utilise la syntaxe

```
gsave
fill
grestore
```

`currentpoint` est une commande qui empile les coordonnées du point courant.

6.3 Opérateurs de contrôle graphique

`setlinewidth` est un opérateur qui nécessite un argument : l'épaisseur du trait à dessiner.

`setgray` est un opérateur qui nécessite un argument réel compris entre 0.0 et 1.0.
0.0 représente la couleur noire et 1.0 la couleur blanche. Entre ces deux valeurs, on obtient tous les niveaux de gris.

`setrgbcolor` est un opérateur nécessitant trois arguments réels compris entre 0.0 et 1.0 ; ces paramètres déterminent les niveaux respectifs de « rouge », « vert » et « bleu ». Il existe également l'opérateur `setcmykcolor` qui nécessite, lui, quatre arguments réels compris entre 0.0 et 1.0 : les niveaux de « cyan », « magenta », « jaune » et « noir ».

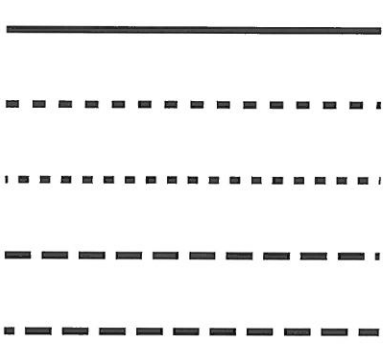
`setdash` est un opérateur nécesitant deux opérateurs : un tableau et un nombre ; il permet de régler le type de pointillé.

Si le tableau est vide, le trait est continu.

Si le tableau comprend un seul nombre, celui-ci représente à la fois le nombre d'unités de longueur du trait et celui de l'intervalle du pointillé.

Si le tableau comprend deux nombres, le premier est le nombre d'unités de longueur du trait et le second, le nombre d'unité de longueur de l'intervalle du pointillé.

Quant au nombre que nécessite cet opérateur, il s'agit d'un *offset* ou déphasage qui permet d'indiquer l'endroit où on commence le dessin du trait à partir du début du *pattern* du pointillé.



`showpage` est la commande qui permet d'imprimer physiquement au moyen du périphérique choisi.

6.4 Autres opérateurs et commandes

`findfont` utilise un argument : la police de caractères ; il place le dictionnaire qui contient la description complète de celle-ci sur la pile d'opérandes.

Les principales polices utilisées sont :

```
/Times-Roman
/Times-Bold
/Times-Italic
/Times-BoldItalic
/Helvetica
/Helvetica-Bold
/Helvetica-Oblique
/Helvetica-BoldOblique
/Courier
/Courier-Bold
/Courier-Oblique
/Courier-BoldOblique
/Symbol
/ZapfDingbats
```

La police `/Symbol` contient notamment les lettres de l'alphabet grec.

La police `/ZapfDingbats` est une police d'icônes.

`scalefont` nécessite un argument qui est la force de corps (exprimée en unités courantes) de la police.

`setfont` est une commande qui établit la police comme police courante dans le contexte graphique.

`show` est un opérateur qui nécessite comme argument une chaîne de caractères qu'il prend sur la pile d'opérandes et transfère dans la mémoire image avec la police courante.
Exemple :

```
/Times-Roman findfont
12 scalefont setfont
100 100 moveto
(Bonjour) show
```

Ce petit programme écrit Bonjour dans la police `/Times-Roman` de force 12 points à partir du point de coordonnées (100, 100). L'unité ici est le point.

`gsave` est une commande qui empile le contexte graphique courant sur la pile des états graphiques (voir page 10).

`grestore` dépile le dernier état graphique empilé par `gsave`.

`%%BoundingBox` définit, par les coordonnées de son coin inférieur gauche et celles de son coin supérieur droit, une boîte qui encadre l'image.
La syntaxe est : `%%BoundingBox: 122 91 274 244`
Cette instruction se positionne en début de programme et l'unité est toujours le point.
Un fichier `PostScript` « pur » possède en général l'extension `.ps` ; un tel fichier adresse une page complète (`A3`, `A4`, ...) Si l'on veut insérer une figure dans un document LaTeX, par exemple, on ne désire certainement pas intégrer la page complète, mais seulement la partie qui contient la figure. On définit alors la *BoundingBox* qui l'encadre et on obtient ainsi un fichier au format `encapsulated PostScript` qui possède l'extension `.eps`.

`%!PS-Adobe-2.0 EPSF-1.2` est une ligne (presque) inutile. Les deux premiers caractères ne sont nécessaires que si l'on travaille avec le logiciel `JustText`. Néanmoins on a l'habitude de conserver cette ligne qui indique la version de `PostScript` utilisée.

6.5 Graphiques de fonctions

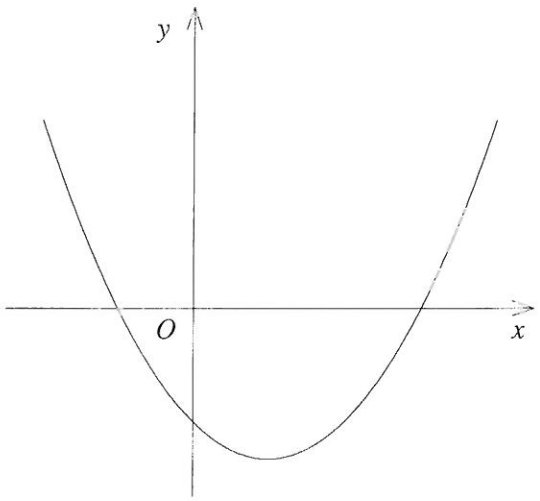
6.5.1 Introduction

Le graphique d'une fonction est un chemin `PostScript` composé d'un ensemble de petits segments – en nombre fini, bien sûr – joignant deux points successifs $(x_i, f(x_i))$ et $(x_{i+1}, f(x_{i+1}))$. C'est l'utilisateur qui détermine le nombre et le choix des x_i. Il est évidemment inutile d'en caculer 2 000 s'il n'y a que 200 points – au sens `PostScript`, c'est-à-dire 200 fois $1/72^{\text{ème}}$ de pouce – dans le domaine sur lequel on veut dessiner le graphique. Pour avoir une idée de l'ordre de grandeur, remarquons que 200 points représentent environ 7.056 cm. On obtient bien sûr la meilleure résolution en faisant varier x de point en point.

6.5.2 Fonction donnée sous la forme $y = f(x)$

Voici un programme qui permet de représenter n'importe quelle parabole.

```
/@binf -2 def
/@bsup 4 def

/a 0.5 def
/b -1 def
/c -1.5 def

/fx {@xc dup mul a mul
@xc b mul add c add} def

/graphique {/@xc @xc 1 s div add def
@xc fx lineto} def

newpath
/@xc @binf def
@binf fx moveto

@bsup @binf sub s mul round cvi
{graphique} repeat

stroke
```



Les deux premières lignes communiquent au logiciel entre quelles valeurs de x le graphique doit être dessiné (l'unité choisie ici est le cm).

Les trois lignes suivantes permettent d'introduire les coefficients a, b, c de la fonction $y = ax^2 + bx + c$.

Les deux lignes qui suivent indiquent comment calculer la valeur de la fonction pour une valeur donnée de x.

Les deux lignes suivantes définissent la manière de tracer le graphique : on incrémente l'abscisse de 1 point (`PostScript`) puis on calcule la valeur de la fonction pour cette nouvelle abscisse et on joint le point courant au point dont on vient de calculer les coordonnées.

Ensuite, on définit un nouveau chemin et on initialise la valeur de x à la borne gauche du domaine sur lequel on dessine.

La ligne `@binf fx moveto` place le point courant à l'endroit du premier point qui sera représenté.

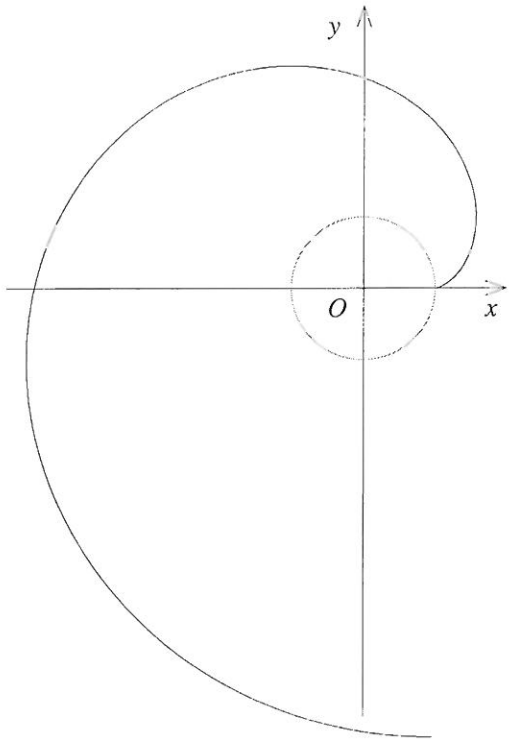
Enfin, la ligne `@bsup @binf sub s mul round cvi` calcule le nombre de fois qu'il faut répéter, dans la boucle `repeat`, la procédure `graphique` qui dessine les petits segments.

6.5.3 Fonction donnée sous forme paramétrique $(x(t), y(t))$

Les mêmes principes ont inspiré l'écriture du programme ci-dessous qui dessine une développante de cercle dont l'équation est, pour un cercle de rayon r :

$$\begin{cases} x &= r\cos t + rt\sin t \\ y &= r\sin t - rt\cos t. \end{cases}$$

```
/pi 3.1415926535 def
/r 1 def

/@binf 0 def
/@bsup pi 2 mul def

/@xt {@tc pi div 180 mul cos r mul
@tc r mul @tc pi div 180 mul sin
mul add} def
/@yt {@tc pi div 180 mul sin r mul
@tc r mul @tc pi div 180 mul cos
mul sub} def

/graphique {/@tc @tc 1 s div add def
@xt @yt lineto} def

newpath
/@tc @binf def
@xt @yt moveto
@bsup @binf sub s mul round cvi
{graphique} repeat
stroke
```



7 Caractères accentués

En ce qui concerne l'écriture des chaînes, le langage `Postscript` se sert de jeux de caractères. L'encodage d'une police de caractères se fait dans un dictionnaire qui contient la description des caractères avec des valeurs associées comprises entre 0 et 255. Cet encodage n'est pas figé et peut être modifié à tout moment au moyen d'un programme. Cette propriété est utilisée notamment dans le but de disposer des caractères accentués – en français, par exemple.
Par défaut, `PostScript` travaille avec un encodage standard qui ne contient que 118 caractères (alphabet américain). Les descriptions des autres caractères tels que lettres accentuées et symboles existent mais ne sont pas affectées dans ce que l'on appelle le vecteur d'encodage. Les 138 autres caractères disponibles ne sont pas utilisés et contiennent le nom « `.notdef` ». Le programmeur qui désire écrire les caractères accentués de la langue française, par exemple, doit redéfinir un dictionnaire de caractères de façon à pouvoir utiliser ces caractères.
Dans cette première initiation au langage, nous n'entrons pas dans les détails. Nous nous contentons de fournir le programme qui permet d'encoder les caractères accentués du français. Son utilisation est fort simple. Il suffit de recopier les instructions de la page 58 dans tout programme `PostScript` qui doit écrire des lettres accentuées. L'appel à cette « macro » se fait grâce à la ligne d'instruction

```
/Times-Roman /Times-Roman NLSVec  ReEncode
```

qui doit précéder l'habituel appel à la police de caractères

```
/Times-Roman findfont
```

Il va de soi que `/Times-Roman` peut être remplacé par le nom de n'importe quelle autre police.

ANNEXE

Macros PostScript

8 Macros

Les macros décrites ci-dessous sont répertoriées dans l'index en caractères gras. Elles sont téléchargeables sur le site du CREM hébergé par PROFOR. C'est pour cette raison que, dans les commentaires, l'usage des caractères accentués a été évité de façon à permettre une lecture claire tant sur Mac que sur PC.

8.1 Macros arithmétiques

8.1.1 max

```
% Macro max
% 2 arguments a fournir (nombres ou chaines).
% Empile le maximum des deux arguments.

 /max {/@opmx2 exch def
       /@opmx1 exch def
       @opmx1 @opmx2 gt
       {@opmx1} {@opmx2} ifelse} def
```

8.1.2 min

```
% Macro min
% 2 arguments a fournir (nombres ou chaines).
% Empile le minimum des deux arguments.

 /min {/@opmn2 exch def
       /@opmn1 exch def
       @opmn1 @opmn2 lt
       {@opmn1} {@opmn2} ifelse} def
```

8.1.3 pgcd

```
% Macro pgcd
% 2 arguments entiers a fournir.
% Empile le pgcd des deux nombres.

 /pgcd
  {
   /@oppgcd2 exch def
   /@oppgcd1 exch def
     {
      @oppgcd2 0 eq {exit} if
      /@temp @oppgcd1 @oppgcd2 mod def
      /@oppgcd1 @oppgcd2 def
      /@oppgcd2 @temp def
     } loop
  @oppgcd1 abs
```

```
 } def
```

8.1.4 ppcm

```
% Macro ppcm
% 2 arguments entiers a fournir.
% Utilise la macro pgcd.
% Empile le ppcm des deux nombres.

 /ppcm
  {
   /@opppcm2 exch def
   /@opppcm1 exch def
   @opppcm1 dup @opppcm2 dup 4 -1 roll
   pgcd idiv mul abs
  } def
```

8.2 Macros qui calculent

8.2.1 Bissectrice

```
% Macro Bissectrice
% 6 arguments a fournir :
% les coefficients des equations des deux droites
% (forme ax + by + c = 0).
% Empile les coefficients de l'equation d'une bissectrice.

 /Bissectrice
  {
   /@wBS2 exch def /@vBS2 exch def
   /@uBS2 exch def /@wBS1 exch def
   /@vBS1 exch def /@uBS1 exch def

   /@deltaBS1 @uBS1 dup mul
   @vBS1 dup mul add sqrt def
   /@deltaBS2 @uBS2 dup mul
   @vBS2 dup mul add sqrt def

   @uBS1 @deltaBS1 div
   @uBS2 @deltaBS2 div add
   @vBS1 @deltaBS1 div
   @vBS2 @deltaBS2 div add
   @wBS1 @deltaBS1 div
   @wBS2 @deltaBS2 div add
  } def
```

8.2.2 CoeffDroite

```
% Macro CoeffDroite
% 4 arguments a fournir :
% les coordonnees de deux points.
% Empile les coefficients a, b, c de l'equation ax + by + c = 0
% de la droite passant par les deux points.

 /CoeffDroite
  {
   /@yCD2 exch def /@xCD2 exch def
   /@yCD1 exch def /@xCD1 exch def

    @yCD1 @yCD2 sub
    @xCD2 @xCD1 sub
    @xCD1 @yCD2 mul @xCD2 @yCD1 mul sub
  } def
```

8.2.3 DeuxPtsDr

```
% Macro DeuxPtsDr
% 3 arguments a fournir :
% les coefficients a, b, c de l'equation ax + by + c = 0 d'une droite.
% Empile les coordonnees de deux points de cette droite.

 /DeuxPtsDr
  {
   /@cDPD exch def /@bDPD exch def
   /@aDPD exch def
   @aDPD 0 eq
     {0 @cDPD @bDPD div neg
      1 @cDPD @bDPD div neg
     }
     {@bDPD 0 eq
        {@cDPD @aDPD div neg 0
         @cDPD @aDPD div neg 1
        }
        {0 @cDPD @bDPD div neg
         @bDPD @aDPD @cDPD @bDPD
         div add neg
        } ifelse
     } ifelse
  } def
```

8.2.4 Distance

```
% Macro Distance
```

```
% 4 arguments a fournir :
% les coordonnees de deux points.
% Empile la distance euclidienne de ces deux points.

 /Distance
  {/@yDi2 exch def /@xDi2 exch def
   /@yDi1 exch def /@xDi1 exch def

   @xDi2 @xDi1 sub dup mul
   @yDi2 @yDi1 sub dup mul
   add sqrt} def
```

8.2.5 DroiteInterCercle

```
% Macro DroiteInterCercle
% 7 arguments a fournir :
% les coordonnees de deux points de la droite,
% les coordonnees du centre du cercle et son rayon.
% Empile selon le cas 0 ou les coordonnees du point de contact et 1
% ou les coordonnees des points d'intersection et 2.

 /DroiteInterCercle
 {
  /@RDIC exch def
  /@yDIC exch def /@xDIC exch def
  /@yDIC2 exch def /@xDIC2 exch def
  /@yDIC1 exch def /@xDIC1 exch def

  /@alphaDIC @yDIC2 @yDIC1 sub
  @xDIC2 @xDIC1 sub atan def

  /@ordDIC @alphaDIC sin neg @xDIC1
  @xDIC sub mul @alphaDIC cos
  @yDIC1 @yDIC sub mul add def

     /@transf
     {
      /@ytransf exch def
      /@xtransf exch def
      @alphaDIC cos @xtransf mul
      @alphaDIC sin @ytransf mul sub
      @xDIC add
      @alphaDIC sin @xtransf mul
      @alphaDIC cos @ytransf mul add
      @yDIC add
     } def
```

```
  @ordDIC @RDIC gt
  @ordDIC @RDIC neg lt xor
     {0}
     {
         @ordDIC @RDIC ne
         @ordDIC @RDIC neg ne and
            {@RDIC dup mul @ordDIC dup mul
             sub sqrt neg @ordDIC @transf
             @RDIC dup mul @ordDIC dup mul
             sub sqrt @ordDIC @transf
             2}
            {
             @ordDIC @RDIC eq
               {0 @RDIC @transf 1}
               {0 @RDIC neg @transf 1} ifelse
            } ifelse
      } ifelse
} def
```

8.2.6 InterDroites

```
% Macro InterDroites
% 8 arguments a fournir :
% les coordonnees de quatre points.
% Chaque paire de points determine une droite.
% Empile les coordonnees du point d'intersection des
% deux droites s'il existe, sinon 0 0.

 /InterDroites
 {
   /@yID4 exch def /@xID4 exch def
   /@yID3 exch def /@xID3 exch def
   /@yID2 exch def /@xID2 exch def
   /@yID1 exch def /@xID1 exch def

   @xID2 @xID1 sub
   @yID4 @yID3 sub mul
   @xID4 @xID3 sub
   @yID2 @yID1 sub mul eq
   { 0 0 }
   { /@lambdaID @yID3 @yID1 sub
   @xID4 @xID3 sub mul
   @xID3 @xID1 sub
   @yID4 @yID3 sub mul sub
   @yID2 @yID1 sub
```

```
   @xID4 @xID3 sub mul
   @xID2 @xID1 sub
   @yID4 @yID3 sub mul sub div def

   @xID1 @lambdaID @xID2 @xID1
   sub mul add
   @yID1 @lambdaID @yID2 @yID1
   sub mul add }  ifelse
 } def
```

8.2.7 Mediatrice

```
% Macro Mediatrice
% 4 arguments a fournir :
% les coordonnees des deux extremites d'un segment.
% Empile les coordonnees du milieu du segment et celles
% d'un autre point de la mediatrice.

 /Mediatrice
 {
  /@yMed2 exch def
  /@xMed2 exch def
  /@yMed1 exch def
  /@xMed1 exch def
     @xMed1 @xMed2 add 2 div dup
     @yMed1 @yMed2 add 2 div dup
     3 1 roll
     exch @yMed1 @yMed2 sub add
     exch @xMed2 @xMed1 sub add
 } def
```

8.2.8 Milieu

```
% Macro Milieu
% 4 arguments a fournir :
% les coordonnees des deux extremites d'un segment.
% Empile les coordonnees du milieu du segment.

 /Milieu
 {
  /@yMil2 exch def
  /@xMil2 exch def
  /@yMil1 exch def
  /@xMil1 exch def
     @xMil1 @xMil2 add 2 div
     @yMil1 @yMil2 add 2 div
```

```
} def
```

8.2.9 Perp

```
% Macro Perp
% 6 arguments a fournir :
% les coordonnees de deux points d'une droite
% et les coordonnees d'un point quelconque.
% Empile les coordonnees du point donne et d'un deuxieme point
% de la perpendiculaire du point donne a la droite donnee.

/Perp
{
 /@yPerp3 exch def
 /@xPerp3 exch def
 /@yPerp2 exch def
 /@xPerp2 exch def
 /@yPerp1 exch def
 /@xPerp1 exch def
    @xPerp3 @yPerp3
    @xPerp3 @yPerp1 add @yPerp2 sub
    @yPerp3 @xPerp2 add @xPerp1 sub
} def
```

8.3 Macros qui dessinent

8.3.1 Carre

```
% Macro Carre
% 4 arguments a fournir :
% les coordonnees de deux sommets consecutifs (sens trigonometrique).
% Trace le carre.

/Carre
 {
  /@yC2 exch def
  /@xC2 exch def
  /@yC1 exch def
  /@xC1 exch def

  gsave
  newpath
  @xC1 @yC1 translate
     4 {0 0 moveto
        @xC2 @xC1 sub
        @yC2 @yC1 sub lineto
        currentpoint translate
```

```
        90 rotate} repeat
   stroke
   grestore
  } def
```

8.3.2 Fleche

```
% Macro Fleche
% 5 arguments a fournir :
% les coordonnees de l'origine de la fleche,
% les coordonnees de son extremite et la longueur de la pointe.
% La macro dessine la fleche.

 /Fleche
  {
   /@lFL exch def
   /@yextrFL exch def     /@xextrFL exch def
   /@yorFL exch def       /@xorFL exch def
   newpath
   @xorFL @yorFL moveto
   @xextrFL @yextrFL lineto stroke
   gsave
   @xextrFL @yextrFL translate
   @yextrFL @yorFL sub
   @xextrFL @xorFL sub atan rotate
   @lFL neg @lFL 3 div neg moveto
   @lFL @lFL 3 div rlineto
   @lFL neg @lFL 3 div rlineto stroke
   @lFL @lFL 3 div neg moveto
   grestore
 } def


% Macro FlecheN
% 5 arguments a fournir :
% les coordonnees de l'origine  de la fleche,
% les coordonnees de son extremite et la longueur de la pointe.
% La macro dessine la fleche noire.

 /FlecheN
  {
   /@lFLN exch def
   /@yextrFLN exch def     /@xextrFLN exch def
   /@yorFLN exch def       /@xorFLN exch def
   newpath
   @xorFLN @yorFLN moveto
```

```
   @xextrFLN @yextrFLN lineto stroke
   gsave
   @xextrFLN @yextrFLN translate
   @yextrFLN @yorFLN sub
   @xextrFLN @xorFLN sub atan rotate
   newpath
   @1FLN neg @1FLN 3 div neg moveto
   @1FLN @1FLN 3 div rlineto
   @1FLN neg @1FLN 3 div rlineto
   closepath
   fill
   @1FLN @1FLN 3 div neg moveto
   grestore
 } def
```

8.3.3 nGone

```
% Macro nGone
% 5 arguments a fournir :
% les coordonnees de deux sommets consecutifs (sens trigonometrique)
% puis le nombre de cotes du n-gone.
% Trace le n-gone.

 /nGone
  {
    /@nnG exch def
    /@ynG2 exch def  /@xnG2 exch def
    /@ynG1 exch def  /@xnG1 exch def
    /@anglenG 360 @nnG div def

    gsave
    newpath
    @xnG1 @ynG1 translate
    @nnG {0 0 moveto
          @xnG2 @xnG1 sub
          @ynG2 @ynG1 sub lineto
          currentpoint translate
          @anglenG rotate} repeat
    stroke
    grestore
 } def
```

8.3.4 Parallelogramme

```
% Macro Parallelogramme
% 6 arguments a fournir :
% les coordonnees de trois sommets consecutifs (sens trigonometrique).
```

```
% Trace le parallelogramme.

 /Parallelogramme
  {
   /@yPar3 exch def  /@xPar3 exch def
   /@yPar2 exch def  /@xPar2 exch def
   /@yPar1 exch def  /@xPar1 exch def

   newpath
   @xPar1 @yPar1 moveto
   @xPar2 @yPar2 lineto
   @xPar3 @yPar3 lineto
   @xPar1 @xPar2 sub
   @yPar1 @yPar2 sub rlineto
   closepath
   stroke
 } def
```

8.3.5 TracerDroite

```
% Macro TracerDroite
% 6 arguments a fournir : les coordonnees de deux points de la droite,
% les abscisses des extremites gauche et droite.
% Trace la droite passant par les deux points entre les abscisses donnees.
% Si les deux points ont meme abscisse,
% trace la verticale entre les points d'ordonnees -50 et 50.

 /TracerDroite
  {
   /@xTDd exch def    /@xTDg exch def
   /@yTD2 exch def    /@xTD2 exch def
   /@yTD1 exch def    /@xTD1 exch def

     newpath
     @xTD1 @xTD2 eq
     {@xTD1 -50 moveto
     @xTD1 50 lineto}
     {@xTDg dup @xTD1 sub
     @yTD2 @yTD1 sub
     @xTD2 @xTD1 sub div mul
     @yTD1 add moveto
     @xTDd dup @xTD1 sub
     @yTD2 @yTD1 sub
     @xTD2 @xTD1 sub div mul
     @yTD1 add lineto} ifelse
     stroke
```

```
} def
```

8.3.6 TriEq

```
% Macro TriEq
% 4 arguments a fournir :
% les coordonnees de deux sommets d'un triangle equilateral
% (sens trigonometrique).
% Dessine le triangle et empile les coordonnees du troisieme sommet.

/TriEq
 {
  /@yTE2 exch def /@xTE2 exch def
  /@yTE1 exch def /@xTE1 exch def

  /@xTE3 @xTE2 @xTE1 sub def
  /@yTE3 @yTE2 @yTE1 sub def

  @xTE1 @xTE3 2 div @yTE3 3 sqrt
  mul 2 div sub add
  @yTE1 @yTE3 2 div @xTE3 3 sqrt
  mul 2 div add add

  2 copy
  newpath
  moveto
  @xTE1 @yTE1 lineto
  @xTE2 @yTE2 lineto
  closepath
  stroke
} def
```

8.4 Autres macros

8.4.1 Mabsproc

Cette macro fixe certains nombres exprimés en points (épaisseur du trait, force de corps de la police, longueur du pointillé) lors de l'insertion du fichier PostScript dans certains logiciels de traitement de texte tels que VTEX sur PC (sortie PDF) ou TEXture sur Macintosh. Voici quelques exemples d'utilisation de cette procédure :

```
0.5 Mabsproc setlinewidth
[10 Mabsproc 4 Mabsproc] 0 setdash
12 Mabsproc scalefont setfont
```

```
%% Macro Mabsproc
% Fixe l'epaisseur des traits et la taille des lettres
% quels que soient l'echelle et la taille du dessin.
```

```
 /Mabsproc {
        0
        matrix defaultmatrix
        dtransform idtransform
        dup mul exch
        dup mul
        add sqrt
 } bind def
```

8.4.2 Echelle

```
% Echelle
% Permet de travailler en centimetres.

/s 72 2.54 div def
```

8.4.3 Caractères accentués

```
/reencdict 12 dict def
/ReEncode
  {reencdict begin
   /newcodesandnames exch def
   /newfontname exch def
   /basefontname exch def
   /basefontdict basefontname findfont def
   /newfont basefontdict maxlength dict def
   basefontdict
     {exch dup /FID ne
       {dup /Encoding eq
         {exch dup length array copy
          newfont 3 1 roll put }
         {exch newfont 3 1 roll put }
         ifelse
       }
       { pop pop }
       ifelse
     } forall
   newfont /FontName newfontname put
   newcodesandnames aload pop
   newcodesandnames length 2 idiv
     {newfont /Encoding get 3 1 roll put}
     repeat
   newfontname newfont definefont pop
   end
} def
% The following is a definition of the
% National Language characters and codes
```

```
/NLSVec [
 16#C0 /Agrave
 16#C1 /Aacute
 16#C2 /Acircumflex
 16#C3 /Atilde
 16#C4 /Adieresis
 16#C5 /Aring
 16#C6 /AE
 16#C7 /Ccedilla
 16#C8 /Egrave
 16#C9 /Eacute
 16#CA /Ecircumflex
 16#CB /Edieresis
 16#CC /Igrave
 16#CD /Iacute
 16#CE /Icircumflex
 16#CF /Idieresis

 16#D1 /Ntilde
 16#D2 /Ograve
 16#D3 /Oacute
 16#D4 /Ocircumflex
 16#D5 /Otilde
 16#D6 /Odieresis
 16#D9 /Ugrave
 16#DA /Uacute
 16#DB /Ucircumflex
 16#DC /Udieresis
 16#DD /Yacute
 16#DF /germandbls

 16#E0 /agrave
 16#E1 /aacute
 16#E2 /acircumflex
 16#E3 /atilde
 16#E4 /adieresis
 16#E5 /aring
 16#E6 /ae
 16#E7 /ccedilla
 16#E8 /egrave
 16#E9 /eacute
 16#EA /ecircumflex
 16#EB /edieresis
 16#EC /igrave
 16#ED /iacute
```

```
 16#EE /icircumflex
 16#EF /idieresis

 16#F1 /ntilde
 16#F2 /ograve
 16#F3 /oacute
 16#F4 /ocircumflex
 16#F5 /otilde
 16#F6 /odieresis
 16#F9 /ugrave
 16#FA /uacute
 16#FB /ucircumflex
 16#FC /udieresis
 16#FD /yacute
 16#FF /ydieresis
] def
```

Références

[1] EMINET, Bernard-Paul, *Le Livre de PostScript*, P.S.I. Micro Édition, 1987.
ISBN 2-86595-462-5

[2] LISMONT, Luc, *Dessins en PostScript et géométrie analytique*, in *PRATIQUER LA GÉOMÉTRIE – Développement d'outils pédagogiques pour un enseignement de la géométrie accessible à tous*, CREM, 2000 (à paraître).

[3] SMITH, Ross A., *Learning PostScript : A Visual Approach*, Peachpit Press, Berkeley, California, 1990.
ISBN 0-938-151-12-6

[4] WARNOCK, John et GESCHKE, Chuck, *PostScript Language Reference (Adobe Systems Incorporated)*, Addison Wesley, third edition, 1999.
ISBN 0-201-37922-8

Index

Table des matières