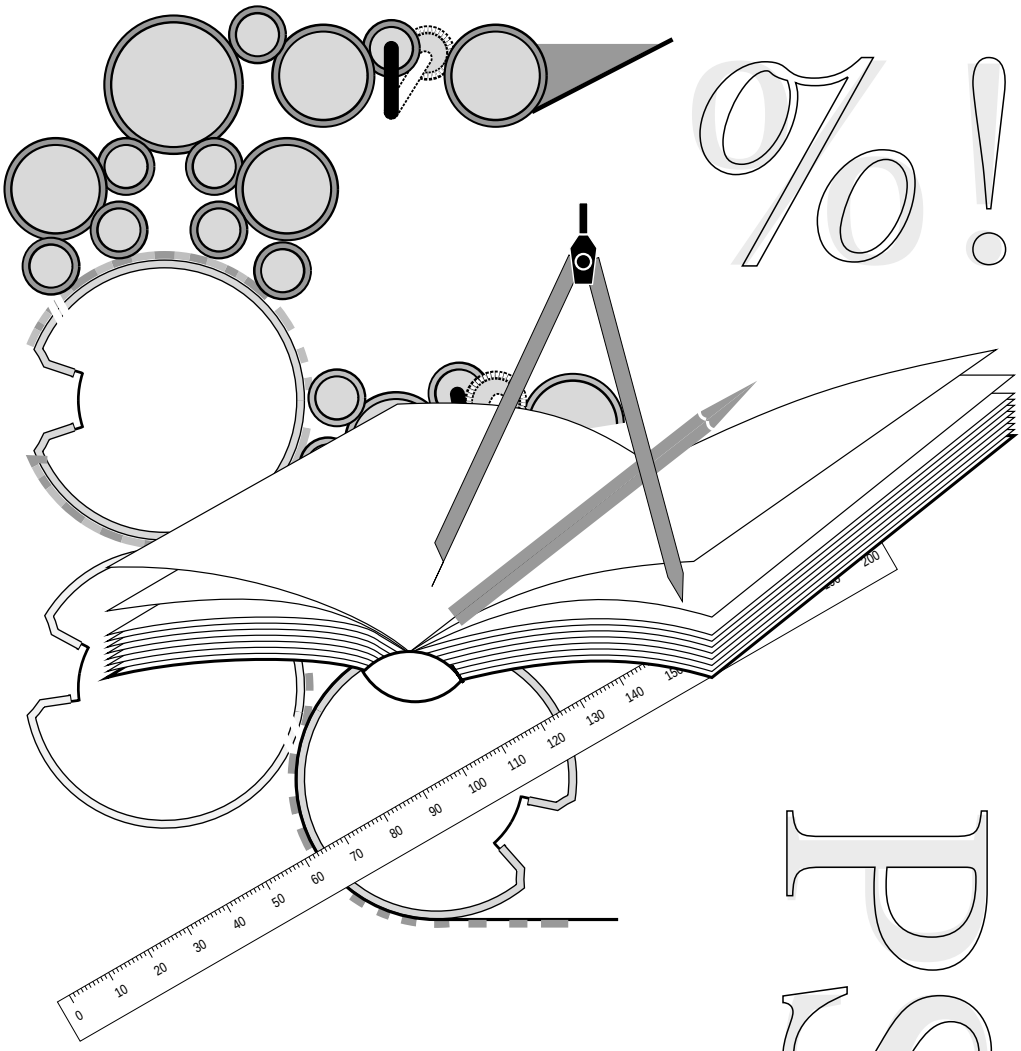
PostScript

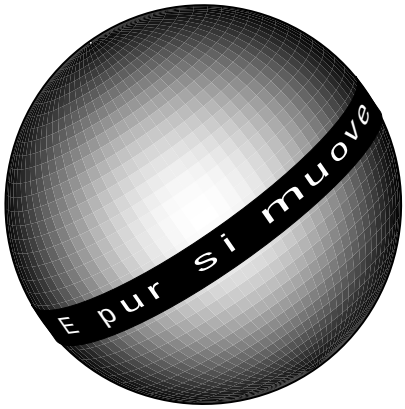
l'essentiel



%!PS

Pierre Blanc

TABLE DES MATIERES



Première édition : 1993
Seconde édition : 1998

AVANT - PROPOS

Le langage PostScript (PS) s'est imposé comme langage de composition de page pour les développeurs de logiciels d'impression. Il permet de coucher sur le papier à peu près tout ce que la typographie moderne est capable de faire. De plus, il autorise des dessins d'excellente qualité et, sur ce point, sert de référence aux meilleurs logiciels graphiques.

A l'origine, PostScript a été conçu pour des développeurs très spécialisés. Mais nombre d'informaticiens ont été séduits par les étonnantes qualités de ce langage et n'ont pas craint d'en explorer les méandres, malgré son abord un peu rébarbatif. On peut prévoir un engouement de plus en plus marqué pour PostScript, car le besoin d'agrémenter les textes par du dessin se fait de plus en plus sentir. Or on assiste depuis peu à deux phénomènes favorables : une baisse continue du prix des imprimantes et surtout la libre disposition d'un excellent interpréteur pour ce langage, à savoir **Ghostscript** associé en général à **GSview.** La programmation directe en PS est devenue tout à fait abordable et d'un coût en tout cas bien inférieur à celui d'un bon logiciel graphique commercial.

Après cette acquisition, le candidat PostScript devra parvenir à la maîtrise du langage, maîtrise qui lui donnera accès à une création de qualité "professionnelle". C'est pour l'aider dans cette quête que ce livre a été écrit. L'auteur a hésité entre une démarche pédagogique qui permettrait de progresser pas à pas vers la connaissance globale et l'écriture d'un ouvrage de référence qui fournirait d'emblée à son utilisateur toutes les ressources disponibles sur chaque aspect de la matière étudiée.

Pour utiliser une image bien connue du lecteur, un ouvrage pédagogique est *à accès série*, alors qu'un livre de référence est à *accès direct (aléatoire* comme disent les professionnels). L'auteur a voulu essayer de concilier l'un et l'autre. Sa démarche sera donc progressive, chaque chapitre nécessitant l'étude des précédents. Cependant, dès le début, seront citées des notions et donnés des exemples de haut niveau, dont la compréhension ne sera totale qu'après initiation. Ces passages seront signalés par une marque spéciale, le trèfle, symbole de la richesse – en PostScript s'entend . Il est donc conseillé d'ignorer en première lecture les passages marqués du symbole ♣ .

A part cette réserve, l'ouvrage ne suppose pas de connaissances spécialisées. L'habitude de la programmation des calculettes, surtout de celles en notation polonaise inverse, des notions de dessin vectoriel en HPGL seront des atouts précieux pour l'étude de PS. Mais rien de tout cela n'est nécessaire. Seuls les éléments de base de l'informatique seront supposés connus. L'auteur ne saurait trop conseiller à un lecteur néophyte de bien vouloir d'abord acquérir ces notions, par exemple dans son manuel *Initiation à l'Informatique* **(1)**. Ces deux livres sont d'ailleurs écrits dans le même style et le même esprit : favoriser l'accès du plus grand nombre aux techniques nouvelles.

Il va sans dire que l'aspect même de cet ouvrage pourra donner une idée sur les capacités de PostScript. Le texte a été saisi avec un logiciel de traitement de texte à sortie optionnelle PostScript (Word-5 sous DOS). Toutes les figures sont générées par des programmes PostScript importés dans des emplacements prévus à cet effet au sein du texte (des *réserves)*. Aucun travail de composition ou de mise en page n'a été nécessaire avant impression. C'est ça PostScript.

Quelques remarques sur le style des caractères employés : **l'écriture en caractères gras** signale bien sûr un passage important. Celle *en italique* attirera l'attention du lecteur sur une expression dont la signification n'est pas courante, mot étranger, mot propre à l'informatique ou à PostScript, ou bien variable mathématique. Les **textes en Helvetica** constituent des exemples d'instructions conformes à PostScript ; si, dans un tel texte, figure un mot en italique, c'est qu'il peut être écrit différemment : il s'agit d'une variable interchangeable avec un autre nom ou un autre objet permis par le langage.

La référence *Adobe* maintes fois citée renverra, pour plus de détails, au livre de base du PostScript **(2)** écrit par la firme Adobe elle-même.

L'auteur tient à remercier les personnes qui l'ont aidé dans sa collecte d'informations sur PostScript, l'imprimerie et l'imagerie informatiques. Ses remerciements s'adressent en particulier à M. Paul Vay, professeur à l'Ecole Supérieure de Papeterie et d'Industries Graphiques, à M. Jean-Marie Crochet, professeur au lycée technique André-Argouges, section imprimerie, à M. Alain Filhol, informaticien à l'Institut Laue-Langevin. Il exprime également sa gratitude à son épouse, Lisette Blanc, qui a bien voulu participer aux illustrations, ainsi qu'à tous ceux qui, d'une manière ou d'une autre, ont apporté leur concours à l'élaboration du présent ouvrage.

---

1 - *Initiation à l'Informatique*, P. Blanc, diffusion Internet.
2 - *PostScript Language Reference Manual*,
Adobe Systems Inc., Addison-Wesley, 1985.

Chapitre 1

INTRODUCTION

1 - UN PEU D'HISTOIRE

Après ses premiers balbutiements, l'informatique a vite cherché à améliorer la présentation des pages qu'elle faisait imprimer. Au début, on ne disposait que d'imprimantes lourdes – à chaînes, à tambour – avec un jeu très réduit de caractères : les majuscules, les chiffres, quelques signes de ponctuation. Cet ensemble suffisait pour *sortir* des résultats de calcul ou des états de gestion rudimentaires, mais leur aspect et leur finition étaient loin d'atteindre la qualité des pages dactylographiées.

Les possibilités de mise en pages étaient réduites. Avec le Fortran, par exemple, on travaillait ligne par ligne. Le premier caractère de la ligne n'était pas imprimé, mais interprété comme caractère de commande, donnant le nombre de lignes d'espacement. Cependant, dès cette époque, des amateurs réussirent à produire des dessins, ou plutôt des tableaux, dont les pixels[1] étaient des caractères ; par exemple, pour simuler des niveaux de gris de plus en plus denses, on utilisait l'espace, le point, le "-", le "I", le "+", le "N", le "M",... et enfin le "W". Pour saisir la beauté de ces dessins, il fallait les regarder d'assez loin. On dit qu'ils possédaient une mauvaise résolution.

Bientôt arrivèrent dans les bureaux des machines de traitement de texte qui, dotées d'un organe de calcul *(processeur)* étaient capables de *justification* et même de *frappe au kilomètre*. Leurs boules ou leurs marguerites, facilement interchangeables, comportant chacune une centaine de caractères, permettaient de progresser vers la qualité de l'impression professionnelle.

L'informatique n'a pu rivaliser avec elles qu'après l'arrivée des imprimantes matricielles. Pour elles, en effet, le caractère n'est plus massif, mais formé de points et donc modifiable à volonté. Certes les premières imprimantes de ce type, avec 8 ou 9 aiguilles donnant autant de points dans la hauteur d'un caractère, n'étaient pas très brillantes, mais des techniques soignées (double passage) permettaient de se rapprocher de la *qualité courrier* ou **LQ** *(letter quality)*, objet de toutes les convoitises. Les imprimantes à 24 aiguilles, puis à jet d'encre, obtinrent encore de meilleurs résultats.

Bientôt l'imprimante matricielle à laser, avec une résolution de 300 points par pouce dépassant le seuil de résolution de l'œil, évinça la machine à écrire, mais posa de nouveaux problèmes : comment gérer les formidables atouts de cette machine ? D'autant plus que cet outil de bureau allait évoluer vers des résolutions supérieures, jusqu'à 1 500 ou 3 000 points par pouce, celle de la photographie, pour aboutir finalement à la *photocomposeuse*, capable de fournir des clichés d'une qualité enfin égale à celle de l'imprimerie.

Les problèmes consistaient à créer des caractères groupés en polices, à les gérer, à les placer sur la page avec toute la souplesse de la typographie classique et à inclure dans cette page des dessins au trait ou des images de type photo, alors que, jusqu'ici, les documents produits dans les bureaux rejetaient les illustrations en fin de volume. Plusieurs groupes de concepteurs travaillèrent sur le sujet dans différentes directions. On inventa les *séquences d'échappement* pour piloter les imprimantes (langage HPLaserJet par exemple), on s'occupa du dessin sur table traçante (langage HPGL), mais les produits les plus perfectionnés visaient plus haut : donner à ces machines les moyens de la typographie professionnelle. Quelques langages y sont parvenu, comme PostScript ou TEX. Ce dernier est très orienté vers l'impression de textes techniques et même scientifiques. Nous ne parlerons dans ce livre que de PostScript.

En 1976, John Warnock conçoit, avec l'aide de John Gaffney, le langage *Design System*. Rejoignant ensuite Xerox au PARC *(Palo Alto Research Center* en Californie), il y développe, avec Martin Newell, le *JaM* et *InterPress*. En 1982, avec Charles Geschke, il crée sa propre société, **Adobe Systems Inc.** qui produira *PostScript*, dans la lignée des précédents. PostScript est une marque de cette société, marque protégée par la législation en vigueur contre la copie et contre tout emploi abusif.

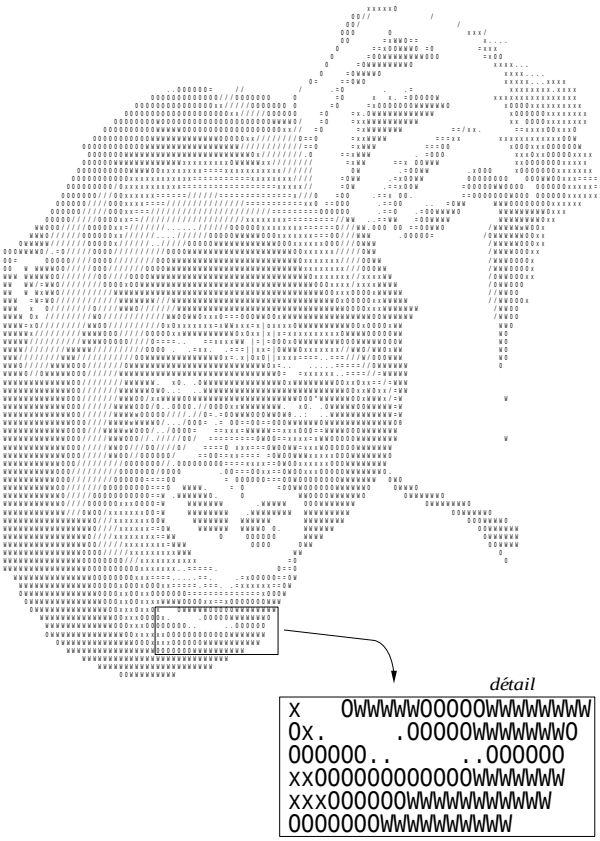
Le langage PostScript a été choisi par Apple pour communiquer avec les imprimantes laser de la série des Macintosh. La qualité des prestations bureautiques de cette gamme d'ordinateurs doit beaucoup à

---

1 - Un pixel est l'élément ultime d'une image, la plus petite surface modifiable.

ce langage. Des machines professionnelles, comme les photocomposeuses Lumonics, ont également adopté PostScript. Le monde IBM y est venu ensuite. Actuellement, les meilleurs logiciels de traitement de texte, de dessin ou de PAO proposent généralement des *sorties* aux normes PostScript, ce qui autorise d'ailleurs entre eux des échanges *(importations)* plus que souhaitables. Côté imprimantes, beaucoup de fabricants proposent au moins en option une carte transformant leur machine en imprimante PostScript.

Avant d'expliquer comment on utilise ce langage, on va définir les principales entités qui le composent. Ce sera l'objet de la suite de ce chapitre.



Gratte-toi la tête, c'est là qu'est le génie ...

Exemple de dessin informatique des années 65-70 (fortement réduit)

2 - PRESENTATION

PostScript est un langage permettant à une imprimante – de bureau ou professionnelle – d'imprimer des documents complexes, illustrations comprises. Il sert à rédiger un *programme* qu'on envoie à ladite imprimante via une console (écran + clavier) ou via un ordinateur.

L'imprimante doit répondre aux *normes PostScript* (en abrégé **PS**). Elle comportera une unité de calcul puissante *(microprocesseur)* associée à un bon volume de mémoire (au moins un mégaoctet), vive et morte. Cette dernière contient un logiciel, l'*interpréteur*, chargé de traiter les instructions du programme et d'organiser le travail de la machine. La mémoire vive conserve temporairement les informations nécessaires *(polices, dictionnaires, états graphiques)*, la fraction d'instructions non encore traitées *(pile)* ainsi que le dessin de la page déjà composée.

Une imprimante ordinaire peut convenir si on dispose d'un interpréteur logiciel comme Ghostscript.

Le programme sera formé d'**instructions** s'exécutant *en séquence* au fur et à mesure de leur arrivée sur la machine. Il n'y a pas de *compilation* (traduction globale) ; c'est un **langage interprété**. Pour faciliter la tâche de l'interpréteur, PS utilise la **notation polonaise inverse** : les opérandes précèdent les opérateurs.

Les instructions ressemblent aux *ordres* des langues humaines, i.e. à des phrases sans sujet, mais avec verbe et compléments directs. Les mots sont délimités par un ou des **espaces**. Aucun signe spécial ne sépare les instructions entre elles. Le verbe s'appelle **opérateur** ou **mot clé** ; c'est le mot fondamental de l'instruction, celui que l'*interpréteur* attend pour l'exécuter, bien que les compléments

soient placés avant lui (ils sont garés provisoirement dans la pile). On peut dire qu'il existe également des compléments circonstanciels; ils sont contenus pour la plupart dans l'*état graphique*, zone de mémoire décrivant comment exécuter les tracés.

Les compléments (des verbes-opérateurs) s'appellent **arguments**, d'un point de vue analyse logique. Vus sous l'aspect grammatical, ce sont des *objets*, lesquels se subdivisent en deux grands groupes :

- les **objets simples**, éléments de PS (nombres et caractères),

- les **objets composites**, qui sont des collections d'objets simples (chaînes, tableaux, fichiers, dictionnaires).

Le langage PS est *procédural*. On écrit des **procédures**, qui peuvent être regardées soit comme des objets, soit comme des opérateurs composites.

3 - NOMBRES ENTIERS

Le nombre entier est l'un des objets les plus simples de PS. Sa gamme d'excursion est normalement de $\pm\ 2^{31} - 1$ ($\pm$ 2 147 483 647). Il s'agit donc d'un entier long de 4 octets **(2)**. Il peut s'exprimer en décimal ou bien dans un système à base *b* sous la forme *b*#*n* pourvu que $2 \leq b \leq 36$. Par exemple :

1992, **16**#7C8, **8**#3710, **2**#11111001000

sont des représentations valides du même nombre, la première forme étant décimale, les suivantes respectivement hexadécimale, octale et binaire.

Les nombres **négatifs** s'écrivent *en décimal* selon la méthode habituelle. Dans les autres bases, c'est plus difficile. Le premier bit est 1 et la représentation est celle du complément à 2. Par exemple le décimal –1 peut s'écrire :

–1, **16**#FFFFFFFF, **8**#37777777777

4 - NOMBRES REELS

Ils ne s'expriment qu'en décimal. On les distingue des entiers par un *point* ou (et) un exposant exprimé sous la forme d'un "**E**" ou d'un "**e**" suivi d'un entier décimal. Le réel peut être positif ou négatif ; sa valeur absolue est comprise entre environ 10^{-38} et 10^{38}.

5 - BOOLEENS

Les booléens sont des entités n'acceptant que deux valeurs, **true** ou **false** *(vrai* ou *faux)*. Selon *Adobe*, ils ne sont pas assimilables, comme dans d'autres langages, aux nombres entiers 1 et 0. Ils sont engendrés comme résultat de comparaisons, de recherches ou d'opérations binaires (**and, not, or, xor, search, read, known**) et servent à aiguiller les options d'opérateurs tels que **if, ifelse, echo, charpath**.

6 - CARACTERES

Selon leur rôle, on peut classer les caractères en quatre catégories :

- **les caractères de programmation**. Ils servent à écrire le programme exécutable. Seuls sont acceptés les caractères du jeu ASCII réduit (du n° 32 au 127 compris), les mêmes que ceux du Pascal ou du C **(3)**.

- **les caractères de commentaires**. On peut ajouter dans un programme, à la suite du caractère spécial **%**, n'importe quel caractère disponible au clavier. Ils seront ignorés par l'interpréteur jusqu'à la fin de la ligne en cours.

- **les caractères à imprimer**. Il s'agit de ceux qu'on désire imprimer. Ils font obligatoirement partie de la police en cours (224 maximum, mais cette police peut varier au cours d'une même page) et constituent les arguments des opérateurs de traitement de chaînes, de polices ou de caractères.

- **les caractères spéciaux**, qui se subdivisent eux-mêmes en deux groupes :

  - les caractères d'encadrement,
  - les séquences d'échappement.

---

2 - Un nombre entier qui dépasserait ces limites serait automatiquement converti en réel.

3 - Majuscules et minuscules sont des caractères différents.

- les **caractères d'encadrement** : *accolades* **{ }**, *parenthèses* **( )**, *crochets* **[ ]**, les deux signes **< >** et les symboles **%** , **/** et ****

{ } encadrent une procédure	**%** annonce un commentaire
( ) encadrent une chaîne	**/** annonce un nom littéral
[ ] encadrent un tableau	**** est le caractère d'échappement
< > encadrent une chaîne hexadécimale.	

- les **séquences d'échappement** représentent un caractère imprimable, non disponible au clavier ou rendu indisponible pour cause de signification spéciale. Précédées du symbole **** (barre inverse ou *backslash)*, ce sont :

\n à la ligne (ASCII 10)	**\f** à la page (ASCII 12)
\r retour-chariot (ASCII 13)	**** barre inverse (ASCII 92)
\b retour arrière (ASCII 8)	**\(** parenthèse ouvrante (ASCII 40)
\t tabulation (ASCII 9)	**\)** parenthèse fermante (ASCII 41)
*****nnn* caractère de code octal *nnn*	*****RC* retour-chariot *(RC)* ignoré

On reconnaît ici les caractères habituels de mise en page. Les symboles ****, **\(** et **\)** permettent d'imprimer la barre inverse et les parenthèses, non disponibles directement au clavier à cause de leur rôle spécial. Les symboles *****nnn* , où *nnn* est un nombre octal d'au plus trois chiffres (de $31_{\mathbf{10}}$ à $256_{\mathbf{10}}$ exclus), désignent le caractère imprimable dont le rang est $nnn_{\mathbf{8}}$ dans la police en cours. Ils sont très employés, spécialement pour désigner les voyelles accentuées, des symboles commerciaux ou mathématiques (leur signification dépend de la police). Mais ils peuvent aussi désigner les caractères courants: ainsi ****141 , dans la plupart des polices, désigne la lettre *a*, mais α dans la police *symbol* et ❁ dans celle appelée *dingbats*.

Enfin, avant un retour-chariot obligé, on peut placer le signe **** qui annihile, pour PS, l'effet du RC . Ce symbole est utile dans une longue chaîne, qu'on désire monoligne, bien qu'elle soit multiligne à la console.

7 - VARIABLES

En PS, comme avec les autres langages informatiques, on peut utiliser des variables. Leur nom ne doit comporter ni caractère spécial **ni espace**. De plus, ce nom doit s'écrire de telle sorte qu'on ne puisse le confondre avec aucun des objets précédents, nombre entier – décimal ou non –, réel ou caractère (ni avec ceux des objets composites *constants* décrits plus loin). Il est fortement recommandé de ne pas lui donner non plus le nom d'un *opérateur* (mais ce n'est pas interdit !).

Par conséquent, tout mot dans le *programme exécutable* (i.e. hors commentaires), non assimilable à un nombre, à un caractère ou à un opérateur, sera interprété comme un nom de *variable* (ou de procédure). Cette variable pourra contenir n'importe quel objet PS, nombre, caractère ou bien – on le verra par la suite – un objet composite, chaîne, tableau, procédure.

La philosophie de ces variables est la même que dans les autres langages, à cela près qu'aucune marque extérieure ne signale le *type* de la variable (comme en Basic) et qu'aucune déclaration de type n'est possible (comme dans la famille Algol). La variable possède cependant un *type* (au sens des autres langages informatiques) : c'est celui de l'objet qu'elle "contient" (i.e. dont la valeur lui a été affectée). Ce type est donc fugitif **(4)**.

Quand on introduit une variable *pour la première fois*, il est **obligatoire** de faire précéder son nom par la barre oblique normale "/", pour signaler à l'interpréteur qu'il ne doit pas chercher la valeur de cet objet, mais en créer un nouveau possédant ce nom. On dit alors parfois qu'il s'agit d'un *littéral*.

8 - OBJET NUL

C'est un objet spécial, avec lequel PS initialise les variables **composites** à leur création, tant qu'aucune valeur ne leur a été affectée. Alors que, dans certains langages, on peut trouver n'importe quoi dans une chaîne ou un tableau non initialisé, ce n'est pas possible en PS : lors de l'emploi éventuel – et inconsidéré – d'une telle variable, PS signalera qu'elle ne contient que des objets **null**. Bien sûr, en cas d'initialisation d'une partie des éléments de cette variable, seuls les éléments non affectés garderont la valeur **null**.

4 - Il est, de plus, peu prononcé. Ainsi, on peut toujours écrire un réel comme un entier (mais non l'inverse).

9 - CHAINES

Les chaînes sont les plus simples des objets composites. Elles sont formées uniquement de **caractères imprimables** ; leur longueur maximale est de 65 535 caractères. Les éléments d'une chaîne sont en réalité pour la machine des valeurs numériques, égales à celles émises par le clavier lors de leur saisie. Leur valeur s'échelonne de 32 à 255 compris.

La chaîne peut s'écrire sous forme d'une suite de *n* caractères encadrés par des parenthèses. On peut dire alors que c'est une *chaîne constante* ou initiale. Les seuls *caractères spéciaux* admis sont ceux avec échappement (donc précédés de ****) ainsi que les parenthèses à condition d'être appariées.
Par exemple :

(*This is a string***)**
(*Tout fuit***\n\r***Tout passe ;* **\n\r***L'espace***\n\r***Efface***\n\r***Le bruit.***)**

sont deux chaînes imprimables avec les polices PS. Les parenthèses n'en font pas partie. La dernière chaîne fera imprimer les cinq vers de Victor Hugo avec des retours en début de ligne précisés par **\r\n** ou **\n\r**). L'expression

(*Ceci est une cha***\214***ne fran***\207***aise (du moins, on l'esp***\212***re !)* **)**

est une chaîne qui sera imprimée (sans les parenthèses externes, mais avec les parenthèses internes) en sautant généralement les trois caractères définis numériquement (**\214**, **\207** et **\212**) car les polices PS, très anglophiles, ne les connaissent pas. On expliquera en fin de volume comment modifier ces polices pour les *franciser* et imprimer le "î" (sous DOS, caractère $140_{10} = 214_8$), le "ç" ($135_{10} = 207_8$) et le "è" ($138_{10} = 212_8$).

On peut également exprimer une chaîne comme une suite de nombres *hexadécimaux* compris entre 0 et 255 (donc comportant chacun deux chiffres de 0 à FF inclus). Cette suite sera encadrée par les signes **<** et **>** . Exemple :

< 49 6E 74 72 6F 64 75 63 74 69 6F 6E **>** équivaut à
(*Introduction***)**

Si on utilise plusieurs fois la même chaîne, il est avantageux (gain de temps de frappe, de temps d'exécution, économie de mémoire) de lui donner un *nom*, donc d'en faire une *variable*, qu'on affectera de sa valeur – une chaîne constante entre **( )** ou **< >** – au moyen des opérateurs décrits pages 24-25.

10 - TABLEAUX

Le tableau permet de grouper sous une même appellation des objets disparates : nombres, caractères, chaînes, dictionnaires ou même autres tableaux. C'est un objet composite *hétérogène*. Cependant, on l'emploie surtout sous une forme homogène. Chaque élément d'un tableau est doté automatiquement par PS d'un indice commençant à 0, dont certains opérateurs utiliseront la valeur (cf. pages 25-26).

Il n'y a pas, à strictement parler, de tableau multidimensionnel, mais rien n'empêche l'utilisateur de créer des tableaux dont les éléments sont des tableaux, dont les éléments sont des tableaux, ...

Un tableau peut être désigné sous un nom global et initialisé avec un tableau constant. Un *tableau constant* s'écrit comme une suite de *n* éléments séparés par des espaces et encadrés par des crochets. Par exemple,

[31 28 31 30 31 30 31 31 30 31 30 31 **]**

est un tableau de 12 entiers donnant le nombre de jours par mois dans une année non bissextile. Un tableau de chaînes s'écrirait comme celui-ci :

[(*Tout fuit,***)(***Tout passe;***)(***L'espace***)(***Efface***)(***Le bruit.***)]**

11 - FICHIERS

PS peut créer, ouvrir et fermer des fichiers dans des volumes de mémoire rattachés à son interpréteur. Il peut les lire et y écrire (voir page 34).

Deux fichiers sont automatiquement créés au début de toute session PS, celui d'entrée *(standard input file)* et celui de sortie *(standard output file)*. La ligne d'entrée de l'imprimante écrit dans celui d'entrée et l'interpréteur PS écrit éventuellement dans celui de sortie (en réponse à des ordres d'écriture ou pour signaler des erreurs). Ce dernier fichier correspond à une éventuelle ligne de sortie (par exemple, les liaisons RS232 et Apple Talk peuvent être utilisées à la fois comme ligne d'entrée et ligne de sortie, mais ce n'est pas le cas de la liaison Centronix, qui est elle monodirectionnelle, cf. page 50).

12 - DICTIONNAIRES

Les dictionnaires sont des tables formées de couples *nom-valeur*. Ils donnent le contenu *(valeur)* de la variable dont le *nom* est cité. L'interpréteur les consulte chaque fois que dans le programme il rencontre un nom de variable ou de procédure.

On peut *créer* des dictionnaires. C'est parfois nécessaire. Ils sont placés dans une pile spéciale, appelée *pile des dictionnaires*. Au début d'une session de travail, le système place en bas de la pile **Systemdict**, qui contient la traduction en langage machine des opérateurs PostScript. Il place au-dessus **Userdict**, qui jouera par défaut le rôle de *dictionnaire courant.* Ces deux dictionnaires ne peuvent être ni déplacés ni retirés de la pile. L'utilisateur peut par contre placer au-dessus d'eux un autre dictionnaire (ou plusieurs) créé par ses soins. Il deviendra dictionnaire courant, celui dans lequel se placeront les couples nom-valeur au moment de leur création (voir page 34).

13 - POLICES

Une police de caractères est un dictionnaire spécial, qui contient, entre autres, 224 couples nom-valeur ou *entrées*. Les noms sont de simples numéros – le rang de classement du caractère – et la valeur une procédure, celle effectuant le tracé du dit caractère. Un caractère PS est donc un dessin : il peut subir toute sorte de transformations, en dimension *(corps)*, orientation, etc. La première construction d'un caractère donné est de ce fait un peu longue, mais le résultat – une image formée de points adaptée à la résolution de l'imprimante – est conservé en *mémoire virtuelle* pour servir à nouveau.

14 - OPERATEURS

Ce sont des entités fournies par PS *(primitives)* pour manipuler les objets, les transformer ou les imprimer. Ils s'appliquent à des opérandes ou arguments, toujours placés avant eux *(notation polonaise)*. Ainsi :

2 3 **add** *toto* **mul**

provoque l'addition de 2 et 3, laisse à leur place 5, puis multiplie le contenu de la variable *toto* par 5. Le résultat restera en place (en haut de la pile, cf. p. 8).

Les procédures peuvent agir comme des opérateurs et manipuler des objets. Mais ce ne sont pas des *primitives*, cataloguées elles dans **Systemdict**.

15 - PROCEDURES

Une procédure PS peut faire penser à la fonction ou au sous-programme dans les autres langages. Elle peut avoir un nom et être initialisée avec une procédure *initiale* faite d'une liste d'opérateurs et d'objets encadrée par deux accolades appariées. Ainsi, dans

```
/sch { getinterval } def      /moy { add 2 div }
def
/triang { moveto 10 0 rlineto 8.66 -5 rlineto
closepath stroke } def
```

l'opérateur **def** , comme son nom l'indique, définit les procédures */sch*, */moy* ou */triang* avec la liste d'instructions entre accolades. Le procédé ressemble à la *définition d'une fonction* ou d'un *sous-programme* dans les autres langages. La première écriture ne fait que redéfinir l'opérateur **getinterval** pour lui donner un nom plus simple, *sch*, procédure qui manipulera les mêmes arguments que l'opérateur **getinterval**. La seconde instruction définit la procédure *moy* , qui effectuera la moyenne de deux nombres (arguments de **add**) devant figurer devant le mot *moy*. Le dernier exemple définit le tracé d'un triangle isocèle avec des côtés longs de 10 unités et débutant à des coordonnées à préciser devant *triang* (ce sont en fait les opérandes de **moveto**). On pourra par la suite écrire :

```
123.5 dup mul   toto toto mul   moy
50 50   triang
```

pour obtenir (par la première instruction) la *moyenne quadratique* entre 123,5 et le contenu de *toto* (**dup** provoque la recopie, la duplication, de l'objet qui le précède). La deuxième instruction fait tracer un triangle isocèle de côté 10 (unités) aux points de coordonnées 50 et 50 (unités).

Tout comme un opérateur, la procédure, si elle exige des arguments, les prélève sur la pile opérationnelle et peut y déposer des objets après exécution.

16 - POINTEURS

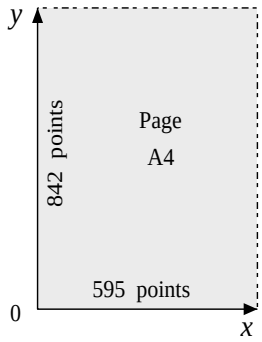
♣ On ne peut quitter les objets composites sans avouer qu'en réalité, ils ne figurent pas *in extenso* dans les dictionnaires comme *valeur* en face du nom qui leur est associé. Ils sont conservés en *mémoire virtuelle* et seule leur adresse (un *pointeur)* figure dans le couple *nom-valeur* du dictionnaire. Cela veut dire que, si plusieurs variables se voient affecter les mêmes objets (avec l'opérateur **copy** par exemple), elles se partageront le *même objet*. Toute modification de l'une pourra entraîner la modification de l'autre.

17 - MEMOIRE VIRTUELLE (VM)

La mémoire virtuelle est en fait de la mémoire vive tout ce qu'il y a de plus réelle. Elle peut être plus ou moins volumineuse selon l'installation ; PS n'en fixe que la valeur maximale (240 K). Un programme peut parfois arriver à la saturer, surtout avec l'emploi de nombreuses polices ou autres objets composites. Dans *Adobe*, il est précisé que chaque nombre d'un tableau ou chaque élément d'une procédure occupe 8 octets (sauf avec le dispositif spécial **setpacking** qui réduirait cette taille à 2,5 octets en moyenne). Chaque caractère occupe un octet, tout couple figurant dans un dictionnaire 20 octets et tout nouveau nom 40 octets, plus autant d'octets que son nom comporte de caractères. Une police exige, elle, de 20 à 30 kilo-octets (en partie à cause de cela, il est recommandé de ne pas utiliser trop de polices différentes ou de corps différents dans un même travail.)

18 - REPERE DES COORDONNEES

Pour exécuter un dessin, il va falloir donner l'emplacement de ses points sur la page. Cet emplacement sera défini par deux coordonnées repérées par un *système* d'axes rectangulaires *(système cartésien)*.



Par défaut, ces axes sont parallèles aux bords du papier, leur origine en est **le coin inférieur gauche** et l'unité de mesure le *point typographique* (parfois appelé **pica**), égal au 1/72e de pouce, donc à 0,35 mm environ (**2,8346 points par mm**). Cependant ce repère est soumis à la matrice CTM, *current transformation matrix*, qui va permettre de le modifier à volonté (cf. p. 20). Une page A4 mesure 595 par 842 points ; mais on peut écrire sur une page de toutes dimensions.

Plus simplement, on peut définir des transformations élémentaires qui modifient la CTM et donc *le repère*, comme des translations, des dilatations, des rotations. Par exemple, une opération très intéressante avant un dessin consiste à redéfinir l'unité des coordonnées avec

soit **2.8346 2.8346 scale**
soit **72 2.54 div dup scale**

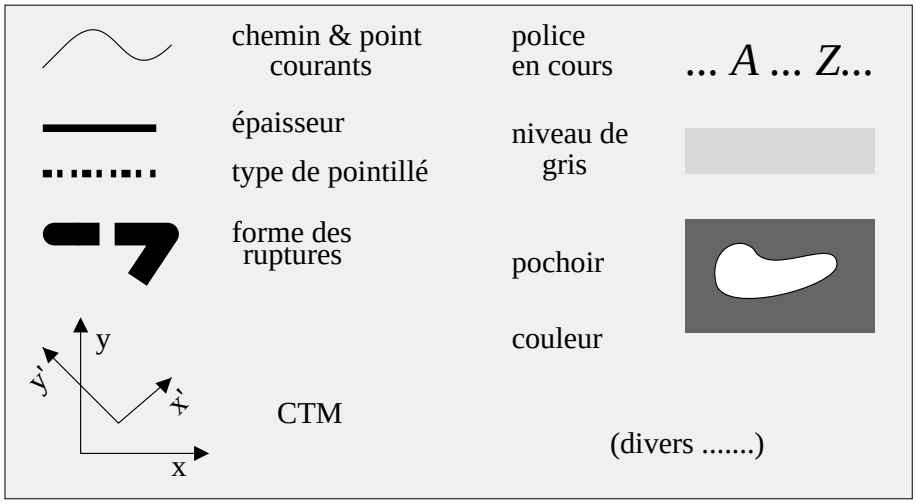
L'une ou l'autre de ces instructions implique qu'il faudra par la suite exprimer en millimètres tous les nombres utilisés comme coordonnées de points.

19 - ETAT GRAPHIQUE

Chaque fois que l'on trace une ligne, il est inutile de préciser un certain nombre de caractéristiques attachées au tracé. L'ensemble de ces caractéristiques s'appelle l'état graphique. Il s'agit de :

- la position courante
- le chemin courant
- l'épaisseur du trait
- la forme des pointillés
- le niveau de gris
- la couleur
- la police et son corps
- le pochoir
- la forme des fins de ligne
- la forme des angles de lignes
- la matrice CTM.

Ces caractéristiques **s'imposent** aux tracés avec la valeur qu'elles possèdent au moment du dit tracé. Mais – et c'est là un des atouts majeurs de PS – on peut sauvegarder en mémoire un état graphique donné, travailler avec un autre, puis récupérer le précédent par une opération très simple.



Principaux éléments d'un état graphique

20 - PILES

La *machinerie* PS (terme utilisé par *Adobe)* manipule trois piles, celle des **dictionnaires** (page 34), celle des **états graphiques** et la **pile opérationnelle.** Toutes ces piles sont du type LIFO *(last in, first out, dernier entré, premier sorti)*. L'objet nouvellement rangé se place en haut de la pile et c'est lui que l'on récupérera en premier à moins de manipulations spéciales (page 33).

Lorsqu'on sauvegarde un *état graphique* (opérateur **gsave**), en réalité on l'empile dans la pile des états graphiques. On pourra modifier alors à son aise l'état graphique courant pour les besoins d'une partie du dessin, puis récupérer facilement l'état antérieur (opérateur **grestore**). Ce procédé est extrêmement utilisé. Noter que cette façon de faire peut consommer beaucoup de mémoire VM si on oublie de *dépiler* (récupérer) des états sauvegardés en grand nombre.

La **pile opérationnelle** ou *pile* tout court est encore plus employée. PS y place tout nom, tout objet du programme autre que les opérateurs. Les *données* s'y accumulent et, lorsqu'un opérateur se présente, il vient les chercher pour en faire les opérandes dont il a besoin. Il laisse les autres en place. Si la pile ne contient pas au moins le nombre d'objets que l'opérateur exige, il y aura erreur par atteinte du bas de pile *(stack underflow)*. Selon l'interpréteur (cf. page 52), ou bien la totalité de la page sera rejetée, ou bien la ligne en cours seule sera abandonnée et l'interpréteur passera à la ligne suivante.

Beaucoup d'opérateurs – et de procédures – calculent ou produisent des valeurs. Ces valeurs se placent en haut de la pile. Ainsi, l'opérateur **add** exige deux nombres qu'il prélève sur le haut de la pile (qu'il *dépile)* et il laisse à leur place un nouveau nombre, égal à la somme des deux précédents. L'opérateur **moveto** *(se déplacer en ...)* exige également deux nombres – qui seront ôtés de la pile – mais n'en place (n'en *empile)* aucun.

Cette pile est tout à fait identique à celle présente dans les calculettes Hewlett-Packard. On peut illustrer son fonctionnement avec une opération mathématique ; par exemple, l'expression algébrique

$$\sin(ax+b)\,\exp(-b/2x)$$

peut s'écrire en PS :

b a x **mul add sin** *b* **2 div** *x* **div neg exp mul**

Il est absolument indispensable d'avoir bien saisi le fonctionnement de cette pile avant de commencer à programmer en PostScript.

Chapitre 2

DROITES ET COURBES

Nous allons aborder dans ce chapitre la manière de réaliser des dessins élémentaires *au trait*, c'est-à-dire de tracer des lignes droites ou courbes et, éventuellement, si ces lignes sont fermées, de remplir d'un grisé ou d'une couleur les surfaces ainsi définies. Pour ce faire, PostScript utilise la notion de chemin *(path)* qu'il est nécessaire de bien comprendre.

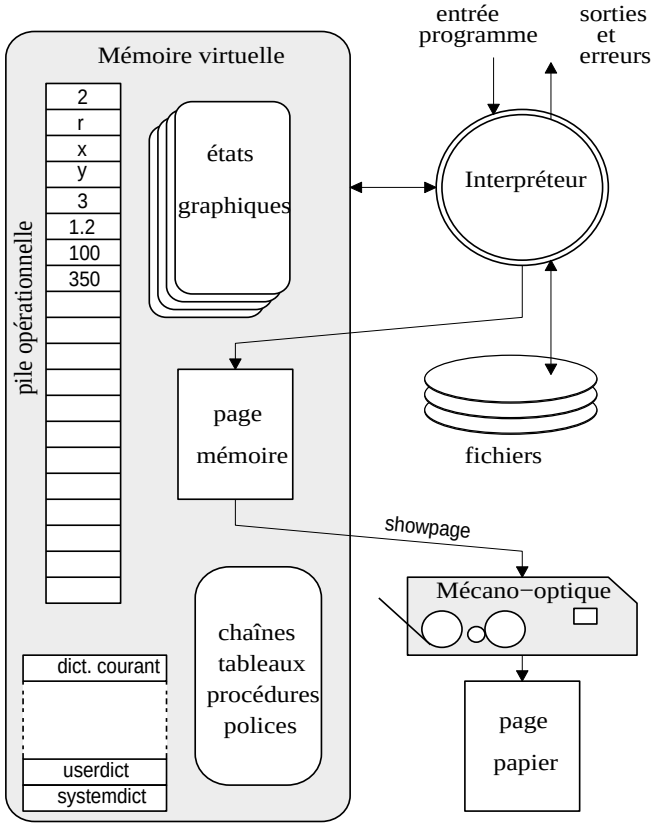
1 - NOTION DE CHEMIN

Le tracé d'un dessin en PS s'opère en trois étapes.

La première consiste à définir un *chemin*, ligne imaginaire ou **théorique**, passant par des points définis par leurs coordonnées (nombres entiers ou réels). Il faudra tout d'abord **créer ce chemin,** l'*ouvrir* si l'on veut, pour lui donner une existence (opérateurs **moveto** ou **rmoveto**). Puis on lui ajoutera, grâce surtout aux opérateurs décrits dans les sections 3 à 8 comprises, des **tronçons**, droits ou courbes, mais toujours **théoriques**.

La seconde étape permettra de colorier ce chemin, de le charger d'encre (de *l'encrer)* – de le matérialiser en quelque sorte – par des opérateurs de *noircissement* (**stroke**, **fill**) qui donneront une épaisseur, une *consistance* aux lignes théoriques. En réalité, à ce stade le tracé est encore immatériel. Il ne fait que s'ajouter dans la mémoire VM aux tracés déjà calculés pour la page en cours. Cette *image de page* est maintenant faite de points qui tiennent compte des caractéristiques de l'imprimante, en particulier de sa résolution.

La troisième étape enfin, avec les opérateurs **showpage** ou **copypage**, actionnera la mécanique de l'imprimante, déposera sur le tambour photosensible l'image-mémoire de la page et la fixera sur la feuille de papier, qui sera ensuite éjectée de la machine.



La machinerie PS

2 - DEPLACEMENT. CREATION D'UN CHEMIN

L'opération consistant à créer un nouveau chemin peut être assimilée à un déplacement de "plume" sur la page, sans tracé (c'est le déplacement *plume haute* des langages pour tables traçantes). Les instructions

x y **moveto** ou *dx dy* **rmoveto**

initialisent un nouveau chemin commençant au point *(x,y)* avec **moveto** et en (x_0+dx, y_0+dy) avec **rmoveto**, mais dans ce cas à condition qu'un chemin ait déjà été ouvert, le point courant étant alors en (x_0, y_0).

Après exécution de (**r**)**moveto**, le point donné devient **point courant.**

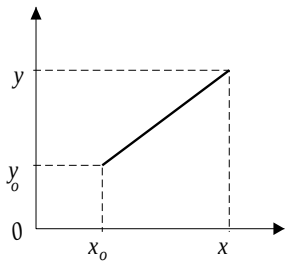
♣ Si un chemin a déjà été ouvert sans avoir reçu aucun tronçon, **moveto** et **rmoveto** provoquent son abandon : il restera ignoré. S'il a reçu des tronçons sans avoir été détruit, il devient *sous-chemin* du chemin en cours (cf. § 10).

3 - SEGMENTS DE DROITE

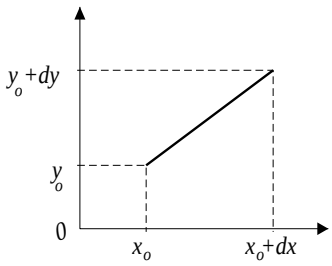
Les deux ordres ci-dessous ajoutent au chemin existant un tronçon égal à un segment de droite commençant au point courant (x_0, y_0) et se terminant au point *(x,y)* avec **lineto**, ou au point (x_0+dx, y_0+dy) avec **rlineto**.

x y **lineto** ou *dx dy* **rlineto**

On aura compris que le préfixe **r** signifie *relatif*. Après exécution de cet ordre, le point atteint devient point courant.



x y **lineto**



dx dy **rlineto**

4 - ARCS DE CERCLE

Trois opérateurs sont envisageables. Avec les deux premiers, **arc** et **arcn**, il faut donner les coordonnées (*x,y*) du centre et le rayon *r* du cercle dont on va prélever un arc sous-tendu par les angles α_1 et α_2 . On écrira :

x y r α_1 α_2 **arc**

ou bien *x y r* α_1 α_2 **arcn**

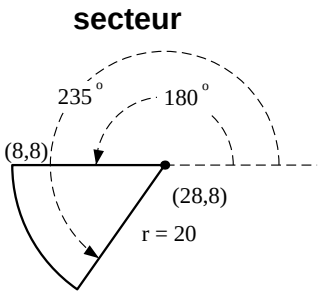
où α_1 et α_2 sont comptés en degrés selon les conventions trigonométriques : origine au point 3 heures des horloges et sens direct, c'est-à-dire contraire à celui des aiguilles d'une montre. L'opérateur **arc** décrit le cercle dans le sens direct, tandis que **arcn** le décrit dans le sens des horloges.

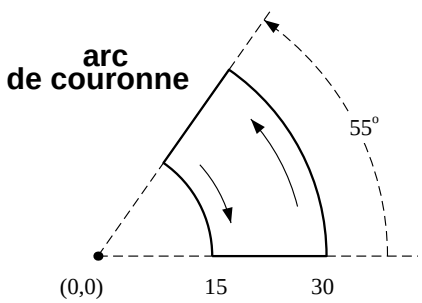
Attention : PS ajoutera au chemin un segment de droite qui part du point courant (x_0, y_0) et rejoint le début de l'arc, c'est-à-dire à l'angle α_1 (figure ci-dessous à droite). Le nouveau point courant est celui correspondant à α_2 . Par exemple :

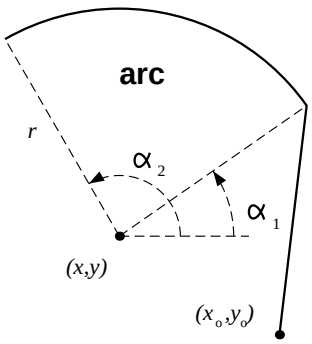
8 8 moveto 28 8 20 180 235 arc closepath

trace un **secteur** (tranche de camembert) centré en 28,8, rayon 20, depuis α_1 = 180° jusqu'à α_2 = 235°., Un **arc de couronne** de 55° fermé est tracé par :

15 0 moveto 0 0 30 0 55 arc 0 0 15 55 0 arcn





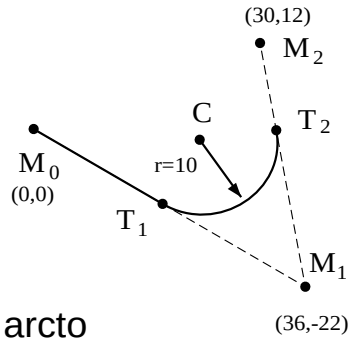


Le troisième opérateur permettant de tracer un arc de cercle est **arcto**. Il exige un point courant (M_0) et 5 arguments : les coordonnées de deux points M_1 et M_2 et un rayon *r*. Cet opérateur trace un arc de cercle de rayon *r* tangent à la fois aux segments M_0M_1 et M_1M_2 (le centre du cercle se trouvera sur la bissectrice de l'angle $M_0M_1M_2$). **arcto** reliera par un segment de droite le point courant M_0 au premier point de tangence T_1 (x_{t1}, y_{t1}) et arrêtera le tracé au deuxième point de tangence T_2 (x_{t2}, y_{t2}), qui devient nouveau point courant. Enfin, **arcto** laisse sur la pile 4 valeurs, les coordonnées de T_1 et de T_2.

$x_1\ y_1\ x_2\ y_2\ r$ **arcto** $\rightarrow\ x_{t1}\ y_{t1}\ x_{t2}\ y_{t2}$

Exemple, pour obtenir la figure ci-contre (*r*=1 cm)

0 0 moveto 36 -22 30 12 10 arcto 4 {pop} repeat



♣ Si on veut réunir un certain nombre de points (x_i, y_i) par des droites dont les angles sont arrondis au rayon r_i, on pourra utiliser la procédure

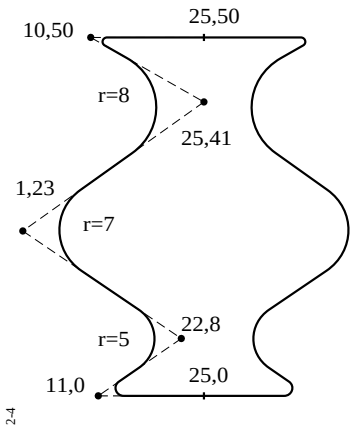
/lignarrond

{ 5 index 5 index 3 -1 roll arcto pop pop pop pop } def

♣ Appliquée à 6 points, pour lesquels on donne les triplets (x_i, y_i, r_i) ,

25 50 moveto 25 0 0 11 0 1.1 22 8 5 1 23 7 25 41 8 10 50 0.6 5 { *lignarrond* **} repeat pop lineto stroke**

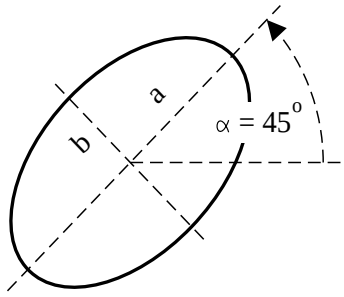
lignarrond dessine la moitié gauche du vase ci-contre. A chaque point (x_i, y_i), elle duplique les valeurs x_{i+1} et y_{i+1} , place leur copie entre y_i et r_i , trace l'arc et détruit les 4 valeurs relatives aux points de tangence. On verra page 20 comment une symétrie permet de dessiner sa partie droite ; on a figuré ici les lignes et cotes ayant servi à construire le vase (ces éléments ne sont évidemment pas dessinés par **arcto**).



5 - ELLIPSES

Pour tracer une ellipse de petit axe 2*b*, de grand axe 2*a* orienté dans la direction α , on peut utiliser une courbe de Bézier (§7), ou tracer un cercle de rayon *a*, après modification du repère de coordonnées (rotation α, contraction en *y* de rapport *b/a*). Pour obtenir l'ellipse ci-contre, centrée en *(x,y)* , de ½ axes *a* = 15, *b* = 15×0.6 et d'orientation α = 45˚, on a écrit :

x y **translate** 45 **rotate** 1 0.6 **scale** 0 15 **rmoveto**
0 0 15 0 360 **arc**



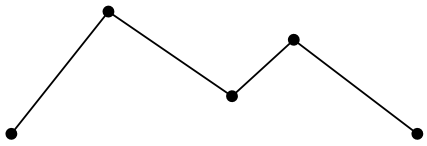
6 - POINTS

Pour figurer un point, on pourrait imprimer avec l'opérateur **show** le caractère *point* de la police en cours. Ce procédé est à déconseiller pour plusieurs raisons (imprécision sur l'emplacement du point, obligation de recourir à différentes polices ...). Il est préférable de tracer avec **arc** un cercle de faible rayon, qu'on remplit de noir avec **fill**. Mais la meilleure façon de dessiner un point consiste à tracer un segment de longueur quasi nulle bénéficiant de l'arrondi final des tracés. Voici une procédure, appelée *point*, qui, précédée de deux coordonnées *x* et *y* , dessine à cette position un petit disque de diamètre *d* (si on a rendu **g** égal à 0 et **setlinecap** à 1, voir p. 18) :

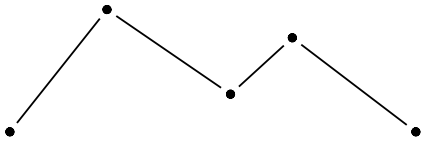
♣ */lpoint* **{moveto** 0 .001 **rlineto gsave** *d* **setlinewidth** *g* **setgray stroke grestore } def**

♣ Par exemple, on tracerait ainsi des lignes brisées :

/*d* 1.5 **def** **/***g* **0 def** *tableau* 2 **copy moveto**
n **{** 2 **copy lineto** 2 **copy stroke** *point* **moveto } repeat**
% figure ci-dessous



[0.01 1.25 1000**] setdash** *tableau* 2 **copy moveto**
n **{**2 **copy lineto** 2 **copy stroke /***g* 1 **def /***d* 2.5 **def**
point 2 **copy /***g* 0 **def /***d* .7 **def** *point* **moveto} repeat**
*tableau désigne une suite de 2*n *nombres, sans [ ].*
% figure ci-dessous



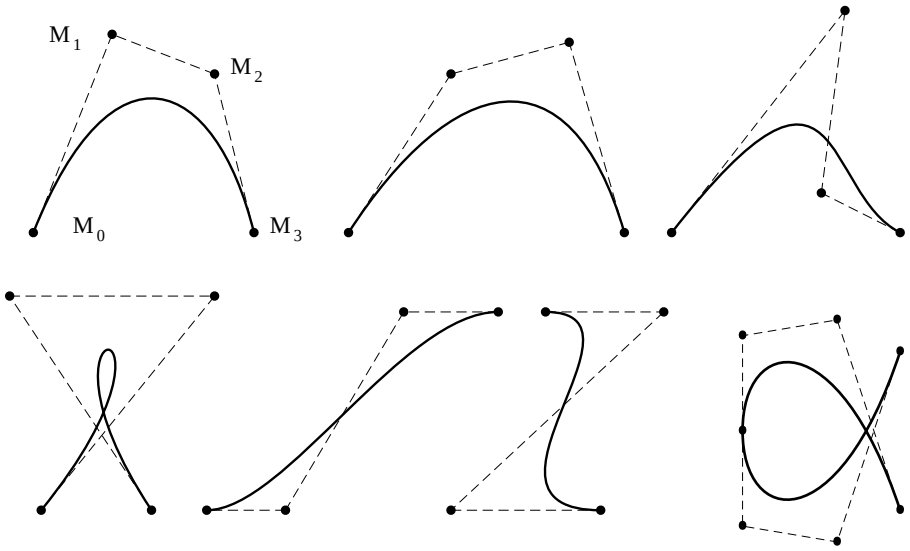
7 - COURBES DE BEZIER

Les courbes de Bézier[1] sont des courbes du 3e degré s'appuyant sur 4 points, le *point courant* M_0 et trois points M_1, M_2, M_3, dont on donnera les coordonnées. La courbe commence au *point courant* M_0 et se termine en M_3 qui devient *point courant*. Elle est tangente en M_0 au segment M_0M_1 et en M_3 au segment M_2M_3. Toujours intérieure au quadrilatère $M_0M_1M_2M_3$, elle reste d'autant mieux "au contact" des segments que ceux-ci sont plus longs. L'instruction suivante ajoute au *chemin courant* une courbe de Bézier commençant en M_0 et s'appuyant sur les points M_0, M_1, M_2, M_3.

x1 y1 x2 y2 x3 y3 curveto

ou bien **dx1 dy1 dx2 dy2 dx3 dy3 rcurveto**

La courbe de Bézier est réversible : on peut la dessiner dans le sens $M_0M_1M_2M_3$ comme dans le sens $M_3M_2M_1M_0$. Autre propriété : ces courbes peuvent directement s'enchaîner, **curveto** ne laissant pas de résidus sur la pile ; le point terminal M_3 devenant point courant, il sera point initial pour la suivante. On rappelle que les points M_0 et M_3 sont des points de contact et de tangence à la courbe alors que M_1 et M_2 ne sont que des points de guidage. On trouvera ci-dessous 6 exemples de courbes de Bézier simples, puis un enchaînement de deux courbes dessinant la lettre α.



8 - RECTANGLES ET POLYGONES

Pour dessiner des figures fermées à côtés rectilignes, on programmera une série de **lineto** ou mieux de **rlineto** . Il est préférable d'éviter de tracer le dernier côté et de s'arrêter au dernier point pour écrire

closepath

qui réunit point courant et point initial (celui du **moveto**). La jonction entre lignes au point initial sera de meilleur aspect[2]. Un rectangle de cotes fixes (20×15) peut se programmer ainsi, *x* et *y* étant connus :

x y moveto 0 20 rlineto 15 0 rlineto 0 –20 rlineto
closepath

1 - Mises au point par Pierre Bézier, ingénieur chez Renault.

2 - Les lignes seront terminées par **linejoin** et non par **linecap** (cf. page 18).

♣ Si plusieurs rectangles sont à tracer, on pourra écrire une procédure, qui sera d'autant plus complexe qu'elle est plus générale. Exemples :

```
/rect  { moveto  0 b rlineto  a 0 rlineto  0 b neg rlineto
         closepath  stroke } def
/a  20  def   /b  15  def     125  286  rect
                          % rectangle b×a en (125,286)
/rec { moveto  /b  exch  def  /a  exch def  0 b  rlineto
  a 0  rlineto  0 b neg  rlineto   closepath  stroke } def
```

Ces procédures tracent un rectangle de largeur *a* et de hauteur *b* en *(x , y)* (coordonnées de son coin inférieur gauche). La dernière s'appelle par :

a b x y rec

♣ Si l'on préfère les rectangles à coins arrondis, on utilisera la procédure *lignarrond*, page 11, ou bien la suivante, un peu plus complexe :

```
/rectar {/r exch def  /b exch def  /a exch def  gsave
  translate  a neg 0  2 copy moveto  a neg b neg  a b
  neg  a b  a neg b
4 { 3 index  3 index  r arcto 4 {pop} repeat } repeat
  a neg 0 lineto  closepath  gsave  g setgray  fill
  grestore  stroke pop pop grestore } def
```

qui dessine le rectangle ci-après, bords arrondis au rayon *r*, largeur 2*a*, hauteur 2*b*, centré en *(x, y)* et ombré avec l'intensité *g*, grâce aux instructions :

/g 0.9 **def** *x y a b r rectar*

9 - TRACE ET REMPLISSAGE

Les opérateurs de définition de tronçons, droits ou courbes, décrits dans les sections 2 à 8 comprises, ne font que définir une **ligne théorique**, un chemin. Pour convertir le chemin en ligne visible, il faut appeler l'opérateur

stroke

Il ne demande pas d'arguments, mais exige l'existence d'un chemin. Il ne dépose aucune valeur sur la pile, mais **détruit le chemin existant**, après l'avoir chargé d'une ligne avec les caractéristiques en cours, c'est-à-dire celles définies dans l'état graphique au moment de l'appel de **stroke** (et non pas celles au moment de l'exécution des opérateurs de chemin). Ces caractéristiques sont :

- l'épaisseur du trait,
- le type de pointillé,
- le type de grisé,
- éventuellement la couleur,
- la forme des extrémités des lignes,
- la forme des points anguleux,
- une limitation pour les angles aigus,

On peut **remplir en grisé** la surface délimitée par un chemin, grâce à l'opérateur

fill

La densité du gris est celle de l'état graphique (voir page 19). Si le chemin n'est pas fermé, **fill** le fermera de la façon qu'il jugera la meilleure ; cela veut dire qu'il vaut mieux fermer le chemin soi-même (avec **closepath**). De plus, après avoir ombré sa surface, **fill** détruit le chemin en cours .

Il y a donc conflit apparent entre **fill** et **stroke**, puisqu'on ne peut pas facilement ni entourer une surface après l'avoir ombrée, ni l'ombrer (ou la colorer) après l'avoir entourée. Ce problème se résout par la sauvegarde temporaire de l'état graphique, qui conserve entre autres le chemin courant.

```
0 0 moveto ...( chemin ) ...  closepath
gsave  0.9  setgray  fill  grestore  stroke
```

Cette séquence permet de remplir la surface avec un gris d'intensité 0.9 en préservant le chemin en cours, puis de marquer ce chemin (ou, si l'on veut, entourer la surface) avec un trait de la couleur courante. Si cette couleur était la même que celle définie pour le **fill**, cette séquence n'aurait aucune utilité. De même, une séquence où le **fill** et le **stroke** seraient permutés, bien que correcte, ne produirait guère d'effet pratique, l'effet du **fill** surchargeant en général celui du **stroke** (y compris sur la frontière du dessin).

10 - CHEMINS COMPLEXES

♣ On a vu qu'un chemin peut comporter plusieurs branches non connectées. On parle alors de **sous-chemins** (*subpath*). De même, un chemin unique peut se couper lui-même et définir des courbes à points multiples. Ce sont deux types différents de chemins complexes.

Le **premier type de complexité** survient si on ouvre un *chemin*, si on lui accole des *tronçons* sans le *terminer* par un opérateur **stroke**, **fill** ou **newpath** (ou en les utilisant entre **gsave** et **grestore**), et si on ouvre ensuite un nouveau chemin. On parle alors de **chemin à branches multiples**. Lorsque l'opérateur **stroke** sera

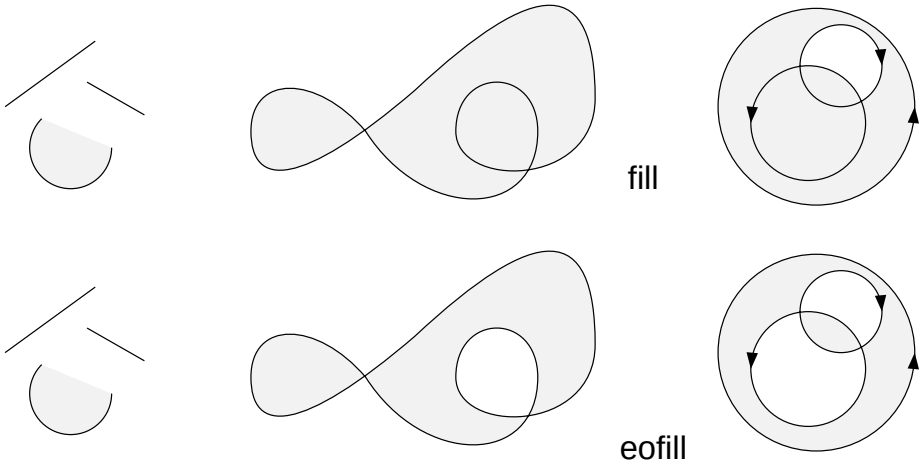
appelé, il encrera et détruira toutes les branches du chemin courant. L'application de l'opérateur **fill** à ce genre de chemin complexe pourra provoquer des surprises, cet opérateur tentant de définir une surface fermée à l'aide des *sous-chemins*. De plus, toute partie ouverte de ces branches, déjà marquée par **stroke**, et contenue dans la surface ombrée par l'opérateur **fill**, sera surchargée par le grisé et disparaîtra. Il faut donc, dans ce cas comme bien souvent, appeler **fill** avant **stroke** et le faire agir entre **gsave** et **grestore.**

Le **deuxième type de chemin complexe** est plus intéressant (chemin unique à *points multiples*). L'opérateur **stroke** n'attire dans ce cas aucune remarque. Il encre tout le chemin, complexe ou pas. Par contre, le remplissage par **fill** pose problème. La surface se subdivise maintenant en plusieurs zones parmi lesquelles il va falloir pratiquer une discrimination, selon qu'on les considère comme extérieures ou intérieures à la surface la plus externe.

L'opérateur **fill** remplit ou non la zone selon la règle du *nombre non nul d'entrelacs*. Une zone est enfermée dans les autres comme le sont des boîtes entre elles. Toute frontière de zone possède une orientation, son sens de tracé. Pour savoir si une zone est considérée comme intérieure ou extérieure, on part d'un point quelconque de la zone et on chemine vers l'extérieur en évitant les points singuliers. On coupe ainsi un certain nombre de frontières, dont on relève le sens. Si on franchit autant de frontières de sens direct que de sens rétrograde, la zone est réputée extérieure (**fill** ne la remplit pas), sinon elle est intérieure.

Pour remplir de telles surfaces, on peut également employer l'opérateur **eofill**, qui obéit à une règle différente, ignorant le sens avec lequel ont été tracées les courbes. La zone ne sera pas remplie si, en partant d'un de ses points et en se dirigeant vers l'extérieur (hors des points singuliers), on franchit un **nombre pair** de frontières (règle de *pair-impair*, *even-odd*). Sinon, elle est remplie.

Ces questions ont l'air académique ; elles sont pourtant très importantes si le dessin est un peu complexe et se posent par exemple dans le tracé des caractères. Les figures suivantes illustrent le comportement de **fill** et de **eofill** dans le cas de chemins complexes.



L'opérateur **closepath** ne réduit pas la complexité éventuelle d'un chemin. Toutes les fois où l'on fait appel à un opérateur réclamant une surface (**fill**, **eofill**, **clip** ...) alors que le chemin n'est pas fermé, PostScript le ferme implicitement comme le ferait **closepath** .

11 - PARTIES CACHEES

Le masquage de certaines parties du dessin est fondamental dès qu'on aspire à représenter la troisième dimension. En effet, les plans lointains doivent être masqués par les objets plus rapprochés.

PS possède l'opérateur **clip** , qui interdit aux fonctions d'encrage-peinture (**stroke**, **fill** ...) d'agir hors des zones autorisées. Il n'interdit pas en revanche d'y définir des chemins. On peut dire qu'il impose un *filtre*, un *pochoir*, pour rappeler qu'il *laisse passer* l'encre. Dans les autres ouvrages, **clip** est souvent traduit par *masque*, ce qui reflète mal la réalité.

Ce pochoir fait partie de l'*état graphique*. Au début, il est équivalent à la totalité de la feuille de papier (en particulier, le système ne signale pas d'erreur si on programme un chemin hors page, il considère qu'il est tout simplement hors filtre). Par la suite, on pourra modifier ce pochoir à l'aide des opérateurs **clip** , **eoclip** et **initclip**.

clip

est un opérateur qui ne demande aucun argument, mais exige l'existence d'un chemin, qu'il ne détruira pas. Il produit un nouveau pochoir égal à l'*intersection* de l'ancien et de la surface enfermée par le chemin courant. Le nouveau filtre ne peut donc pas être plus grand que le précédent ; on ne peut pas agrandir un pochoir, sinon par l'un des biais suivants :

- revenir au filtre d'origine par **initclip** (qui détruit le filtre en cours),

- sauvegarder un filtre donné par **gsave**, puis le récupérer avec **grestore**.

Si, au moment du **clip**, le chemin courant est complexe, la règle des entrelacs s'applique. Si l'on veut appliquer la règle des frontières paires-impaires, on utilisera l'opérateur **eoclip** , qui, pour le reste, est identique à **clip** .

Ces opérateurs n'agissent pas sur les dessins déjà tracés, si bien qu'un dessinateur devrait commencer par les premiers plans et utiliser comme filtre pour les plans suivants les parties extérieures aux objets dessinés. Il est possible, plus simplement, de commencer par les plans lointains, puis d'y superposer des objets dont on fera précéder l'encrage (et non le dessin du chemin) par la séquence suivante, qui efface le contenu du chemin courant.

```
gsave clip clippath 1 setgray fill grestore
```

Si l'on veut simuler *la troisième dimension*, cette notion de **clip** n'est pas la plus appropriée. On utilisera plutôt la technique qui consiste à commencer par dessiner les plans éloignés et à y superposer des objets de plus en plus proches.

Dans un premier temps, on fermera le tracé de cet objet, puis on *peindra* – éventuellement en blanc – la surface ainsi définie. Cette peinture effacera en cet endroit le *fond* du dessin. Dans un deuxième temps, si nécessaire, on pourra dessiner des traits dans ce nouvel objet.

La figure ci-dessous représente un empilement de boules, dessiné en commençant par les plus lointaines et en les peignant avec **fill** , sans employer l'opérateur **clip** . Pour tracer une boule, on a écrit :

```
x  y moveto   r 0 rmoveto
x  y  r 0 360 arc  gsave  g fill grestore stroke
```

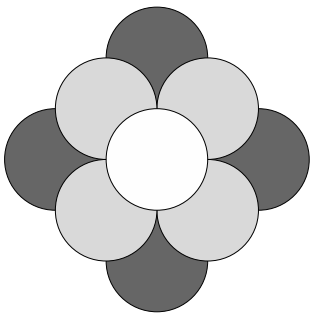
où les variables *x, y, r* et *g* doivent avoir été définies auparavant. On verra p. 32 comment les notions de procédure et de boucle permettent d'automatiser ce dessin.

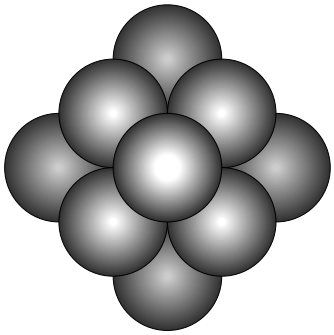
♣ Ces boules seraient plus belles avec un dégradé de gris. En anticipant sur les chapitres 4 (programmation structurée) et 8 (images en demi-teintes), on signale qu'on peut programmer facilement en PS un tel perfectionnement :

```
gsave  2.8346 dup  scale

/cerc  { x y r1 add moveto x y r1 90 450 arc g1 setgray
         fill }  def                    % anneau élémentaire
/boul  {/y exch def  /x exch def
      /dr r 30 div def  /r1 r def  /g1 .1 def  /dg g 25 div def
      29 { cerc /r1 r1 dr sub def /g1 g1 dg add def
      /dg dg 1.02 div def} repeat          % dégradé
      x y r add moveto 0 setgray x y r 90 450 arc stroke
     } def

60  60  translate  3  3  scale
/r 6 def   /g 0.6 def
r 2 mul neg 0 boul  0 r 2 mul boul  0 r 2 mul neg boul
r 2 mul 0 boul                      % les 4 boules basses
/g 0.8 def   r r boul   r neg r boul   r r neg boul
  r neg r neg boul                  % les 4 boules médianes
/g 1 def    0 0 boul   grestore      % boule supérieure
showpage
```





Chapitre 3

L'ETAT GRAPHIQUE

La notion d'état graphique a déjà été esquissée pages 7 et 8. C'est un ensemble de caractéristiques affectant les tracés et les écritures de PostScript. Au début d'un travail (d'une *session)*, ces caractéristiques ont une valeur, dite valeur *par défaut*. On peut bien entendu la modifier par les instructions que nous allons étudier maintenant.

Modifiées ou non, à un instant donné, ces caractéristiques ont une valeur, la *valeur courante* ou *active*. C'est elle qui agit sur les tracés de PostScript, spécialement sur **stroke** et **fill** (ainsi que sur les écritures de texte), au moment où ces opérateurs sont appelés.

1 - CHEMIN ET POINT COURANTS

Même au risque de redite, on ne saurait trop insister sur les notions de chemin et de point courants. Au début de la session, ils ont la valeur **null** . Un *chemin* doit être **créé** ou *ouvert* par (**r**)**moveto** (qui crée également un *point courant)*, puis **construit** par tronçons avec des opérateurs de tracé, tels que ceux étudiés dans le chapitre 2, (**r**)**lineto**, (**r**)**arcto**, **arc**(**n**), (**r**)**curveto**, **closepath** . D'autres opérateurs, liés aux caractères, peuvent également ajouter des tronçons (voir pages 38-39). Le chemin, *théorique* jusque-là, se **concrétise** avec **stroke** ou, dans certains cas, avec **fill** , ces deux opérateurs *détruisant* après action le chemin en cours **(1)**.

Chaque opérateur agissant sur le chemin doit disposer d'un *point courant* dont il modifiera la position. Le point courant est *détruit* en même temps que le chemin courant. Cependant, il existe un opérateur permettant de déposer sur la pile les deux coordonnées x_0 et y_0 du point courant. Il s'agit de

currentpoint

C'est un opérateur très utile. En effet,

> ☞ **l'état graphique en cours s'impose à l'ensemble des tronçons d'un chemin.**

Corollaire : si, dans un même dessin, on veut utiliser des caractéristiques graphiques différentes, il *faut* changer de chemin.

Bien que, du point de vue *état graphique*, on change de chemin, il se peut que *géométriquement* on continue la même figure. Il s'agit alors de raccorder proprement le nouveau chemin au précédent. La meilleure façon est de déposer sur la pile le *point courant* terminant le premier chemin et de s'en servir comme point initial du second. La séquence prend l'allure suivante :

x_0 y_0 **moveto** *... (1er chemin)...* **currentpoint stroke moveto** *... (2e chemin)...*

En effet, si **stroke** et **fill** détruisent chemin et point courants, en aucun cas ils n'agissent sur la pile qui, grâce à **currentpoint** , préserve les coordonnées du point courant pour la poursuite du dessin (voir la remarque importante citée en haut de la page 21).

2 - EPAISSEUR ET FORME DU TRAIT

Ce sont les caractéristiques les plus évidentes attachées à un trait. Plus précisément, il s'agit des caractéristiques ci-contre, dont chacune peut être modifiée comme il va l'être expliqué dans la page suivante :

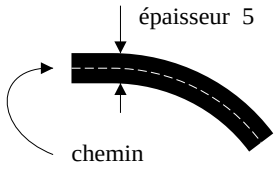
- l'*épaisseur* du trait,
- le motif du *pointillé* (ou *tireté)*,
- le type de *terminaison* des lignes,
- le type des *angles* ou *coudes* de lignes,
- la *limitation* des raccordements aigus.

---

1 - Tout comme **fill** et **stroke**, l'opérateur **newpath** détruit également le chemin courant, sans rien tracer.

- **épaisseur** : *x* **setlinewidth**

qui impose, pour épaisseur des nouveaux traits, la valeur *x* unités, *x* étant un nombre entier ou réel. L'unité, qui dépend de la CTM et de l'opérateur **scale** (voir page 19), a pour valeur par défaut un *point*. La valeur *x*=0 n'est pas recommandée, car elle fournit une épaisseur minimale dépendant de la machine.



- **motif du pointillé** : **[** n_1 n_2 **. . .]** *m* **setdash**

Exemples ci-contre (l'unité est le millimètre) :

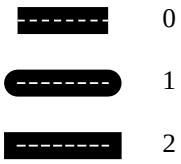
setdash exige deux arguments : un tableau de réels, plus un réel isolé *m*. Le tableau détaille la période du pointillé, le premier nombre donnant la longueur du trait noir, le second celle de l'espacement suivant, etc. Logiquement, ce tableau devrait donc comporter un nombre pair d'éléments, mais ce n'est pas obligatoire. Quant au nombre *m*, c'est une sorte de *décalage*. Il indique quelle longueur de la période doit être "sautée" avant de commencer le tracé de la ligne. Ce décalage est généralement nul. Les éléments du tableau ne peuvent pas être tous égaux à zéro, bien que le tableau, lui, puisse être **null.** Un tableau nul, argument de l'instruction

[ ] 0 **setdash**

provoque, pour les tracés suivants, l'annulation de l'ordre de tracé discontinu et le retour au trait plein **(2)**.

- **terminaison des lignes** : *n* **setlinecap**

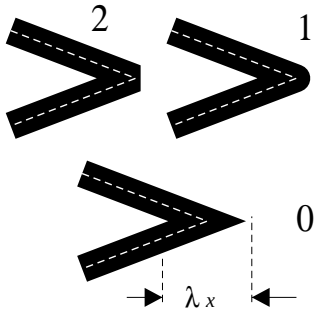
définit la forme de l'extrémité de la ligne (d'épaisseur *x)*



n=0 : arrêt brusque, à angle droit,
n=1 : arrondi avec demi-cercle de diamètre *x*,
n=2 : ajout d'un rectangle de longueur *x*/2 .

- **angle entre lignes** : *n* **setlinejoin**

A la jonction de deux lignes non parallèles, d'épaisseur *x*, l'arête sera, selon la figure ci-contre,



aiguë, si *n* = 0 ,
arrondie au rayon *x*/2, si *n*=1 ,
chanfreinée, si *n*=2 .

- **limitation de jonction** : λ **setmiterlimit**

impose une limite à l'extension des arêtes produits par **setlinejoin** quand l'angle est faible. Au-delà de cette limite, les arêtes seront tronquées. L'argument λ (λ ≥ 1) est égal au rapport entre la longueur du raccord et l'épaisseur *x*, de sorte qu'avec λ=1.4, le chanfrein intervient à partir de 90˚, avec λ=2 à partir de 60˚ et λ=10 (valeur par défaut) à partir de 10˚.

---

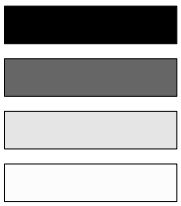
2 - On a déjà utilisé un tracé discontinu page 12. L'argument-tableau était [0.01 1.25 1000]. Il nous a permis de débuter la ligne par un point, puis de *lever la plume* sur 1.25 unités (mm) avant de repartir en trait plein (sur une longueur dépassant celle de la page). Cette instruction détachait de 1.25 mm, côté origine, les lignes issues d'un point. A l'autre extrémité, c'est un petit disque blanc de rayon 1.25 mm qui, en effaçant la fin de ligne, assurait ce détachement.

3 - NIVEAU DE GRIS ET COULEUR

Le niveau de gris joue sur tout tracé, qu'il soit exécuté par **fill , stroke**, ou même **show**. Il se modifie par

g **setgray**

g peut varier de 0 (noir, valeur par défaut) à 1 (blanc, couleur du fond de page).



Le rendu de ce gris dépendra beaucoup de la machine et de sa résolution. *g* est par défaut égal au rapport du nombre de pixels blancs au nombre total de pixels dans la cellule élémentaire d'impression (cf. chap. 8). **setgray** ne modifie jamais la teinte d'une couleur, il ne sert qu'au gris (noir et blanc).

Pour une imprimante couleur, deux opérateurs au choix fournissent la *couleur courante* à partir des trois fondamentales RVB, rouge-vert-bleu. Le plus simple est

r v b **setrgbcolor**

où les 3 variables, comprises entre 0 et 1, donnent la proportion des composantes dans la couleur recherchée.

Le deuxième utilise le *triangle colorimétrique* pour définir la teinte *h (hue)*, la pureté *s (saturation)* et l'intensité *b (brightness)* de la couleur construite, tous nombres compris entre 0 et 1. L'instruction correspondante est

h s b **sethsbcolor**

4 - MODIFICATIONS SIMPLES DES COORDONNEES

Comme il est extrêmement facile en PS de retrouver le repère initial, il est classique de changer très souvent les coordonnées pour dessiner à un autre emplacement ou sous un autre aspect un objet déjà composé. C'est cette facilité qui est mise à profit dans les logiciels de dessin pour placer n'importe où sur la page des copies, agrandies, contractées ou pivotées, de formes élémentaires contenues dans une *boîte à outils*. On va détailler ici les trois principales opérations de ce genre.

- translation

En début de session, l'origine des coordonnées est le coin inférieur gauche de la page (figure page 7). On peut placer cette origine en n'importe quel point, même hors page, grâce à

x y **translate**

x , y étant les coordonnées du nouveau zéro, exprimées en unités courantes.

- rotation

En début de session, le repère des coordonnées est parallèle aux bords de la page. On peut faire pivoter ce repère autour de son origine avec

α **rotate**

où α est l'angle de rotation, exprimé en degrés et compté positivement dans le sens inverse des horloges.

- changement d'échelle

En début de session, l'unité des coordonnées est le point typo ou *pica (dot)* égal à 1/72 pouce *(inch)*. On peut modifier cette échelle en *x* et en *y* grâce à

r_x r_y **scale**

qui multiplie l'unité courante en *x* par r_x et en *y* par r_y Aussi est-il conseillé d'écrire en début de programme

2.8346 2.8346 **scale**
ou bien 2.8346 **dup scale**
ou bien 72 25.4 **div dup scale**

ce qui revient à définir le millimètre comme *unité courante* **(3)**. On rappelle que l'unité courante joue aussi bien sur la position des points, sur l'épaisseur des traits que sur la taille des caractères (figure p. 20). On verra également page 20 que des **scale** avec arguments négatifs provoquent des symétries.

- modifications combinées

Les trois opérateurs précédents modifient en réalité la CTM. Leurs effets ne se font sentir que sur les dessins ou textes créés *après* leur appel. Ils ne modifient en rien les tracés déjà réalisés. Ces opérateurs restent actifs,

- soit jusqu'à nouvel appel du même opérateur,
- soit jusqu'à modification explicite de la CTM,
- soit jusqu'au retour à un état graphique antérieur.

---

3 - Pour tous les exemples de tracé donnés dans cet ouvrage, l'unité courante sera le millimètre.

Leurs effets sont cumulatifs. Ainsi, deux appels successifs **(4)**

$n_1\ n_2$ **translate** $n_3\ n_4$ **translate**
$\Leftrightarrow\ (n_1{+}n_3)\ (n_2{+}n_4)$ **translate**

Deux rotations successives :

α_1 **rotate** α_2 **rotate** $\Leftrightarrow\ (\alpha_1{+}\alpha_2)$ **rotate**

Deux changements d'échelle :

$r_x\ r_y$ **scale** $s_x\ s_y$ **scale** $\Leftrightarrow\ (r_x \times s_x)\ (r_y \times s_y)$ **scale**

Si on veut combiner des opérateurs différents, il faut prendre quelques précautions, les arguments de l'un ou de l'autre pouvant être modifiés par l'opérateur précédent. Il existe un ordre qui met le dessin à l'abri de ces inconvénients, c'est

translate rotate scale

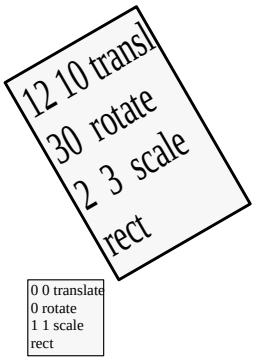
Donnons un exemple. Une procédure *rect* a pu être construite a priori, telle que

/rect { 0 0 **moveto** 1 0 **lineto** 1 1 **lineto**
−1 0 **lineto closepath** } **def**

On a construit en réalité un carré (on aurait pu employer des **rlineto**). Un rectangle de dimensions $a \times b$, orienté selon l'angle α, sera placé dans la page avec son origine en *(x, y)* , grâce à :

gsave *x y* **translate** α **rotate**
a b **scale** *rect* **grestore**

Il est donc justifié de parler en PS d'*objets* qu'on transforme et qu'on place n'importe où dans la page.



Mentionnons un problème pratique courant : comment agrandir d'un facteur *n* la portion de dessin commençant en $x_1\ y_1$ *(zoom)* ? Il faut écrire :

n n **scale** $-x_1\ -y_1$ **translate**

5 - MATRICE DE TRANSFORMATION. SYMETRIES

Dans cet ouvrage, comme d'ailleurs dans *Adobe*, on abrégera en CTM le nom de la matrice de transformation active *(current transformation matrix)*. Cette matrice, d'ordre 3, se ramène à un tableau de 6 valeurs

[*a* α β *b c d*]

tel que les coordonnées dans le repère initial *x'* et *y'* se déduisent des coordonnées du repère actuel *x* et *y* par le système algébrique :

$$\begin{aligned} x' = ax + \beta y + c &= r_x\ x\cos\theta - r_x\ y\sin\theta + c \\ y' = \alpha x + by + d &= r_y\ x\sin\theta + r_y\ y\cos\theta + d \end{aligned}$$

où *c* et *d* sont les valeurs de la translation totale en *x* et *y*, r_x et r_y les facteurs d'échelle (cumulés) et θ l'angle de rotation totale. On peut dire que la CTM convertit les *coordonnées-utilisateur* en *coordonnées-machine*.

On peut modifier directement la CTM par deux opérateurs. Le premier, **setmatrix**, précédé d'un tableau de 6 nombres, remplace la CTM en cours par celle déduite du tableau. Son emploi est cependant déconseillé par *Adobe*. Il vaut mieux procéder par multiplication entre la matrice en cours et celle déduite d'un tableau de 6 nombres grâce à

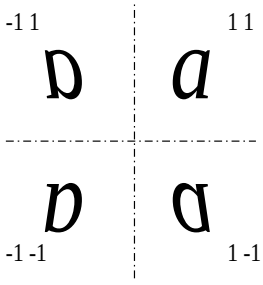
[$n_1\ n_2\ n_3\ n_4\ n_5\ n_6$] **concat**

Par exemple, on dessine des objets symétriques d'un premier avec les transformations suivantes, selon le type de symétrie :

[−1 0 0 1 0 0] **concat** *(Symétrie d'axe vertical)*
[1 0 0 −1 0 0] **concat** *(Symétr. d'axe horizontal)*
[−1 0 0 −1 0 0] **concat** *(Symétrie centrale)*

L'examen de ces matrices montre qu'on peut, beaucoup plus simplement, obtenir les mêmes symétries en utilisant uniquement l'opérateur **scale** avec des arguments négatifs (figure ci-contre) :

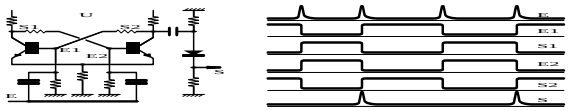
−1 1 **scale** 1 −1 **scale** −1 −1 **scale**



4 - Le symbole ⇔ signifie *équivaut à* . Les écritures $(n_1{+}n_2)$ ... $(r_y{\times}s_y)$ ne sont pas conformes aux règles PS, qui exigeraient d'écrire $n_1\ n_2$ **add** ... $r_y\ s_y$ **mul**.

L'opérateur **concat** permet de déformer des figures. Il peut être utile dans un dessin à 3 dimensions où la valeur de la coordonnée *x'* dépend de celle de *y*. Le coefficient β ne sera donc plus nul. On peut également obtenir ainsi une **écriture penchée**. Par exemple, la figure ci-contre est générée avec les instructions :

[1 0 -.8 1 0 0] concat rec 1.5 .1 M (12345) show



(on a utilisé la police droite Helvetica, cf. chap. 9)

6 - POLICE, POCHOIR ET LES AUTRES

Sont également conservés dans l'état graphique :

- le nom et le *corps* de la *police en cours*, dont on parlera dans le chapitre 7 ;

- le *pochoir* qu'on a étudié page 15 ;

- la *tolérance (flatness)* entre la forme mathématique des courbes dessinées et leur approximation sous forme de segments de droite. C'est la façon dont procède PS pour tracer les courbes. Une tolérance élevée donne des courbes moins lisses, une tolérance faible entraîne des temps d'exécution plus longs. La valeur par défaut est 1, elle est exprimée en pixels ;

- la *fonction de transfert* (**settransfer**) qui permet d'obtenir une sensation visuelle plus progressive dans l'échelle des gris, dépendant beaucoup de l'imprimante. On se rend compte, page 19, de la mauvaise correspondance entre la valeur de l'argument de **setgray** et la sensation de gris. On a dû utiliser des valeurs élevées (0.9, 0.99) pour illustrer notre échelle de gris. L'opérateur **settransfer**, dont l'argument est une procédure, permet de remédier à cet effet. On peut même obtenir, grâce à lui, le négatif d'une page comme par photographie ;

- l'algorithme de rendu des demi-teintes *(halftone pattern, setscreen)*. Cette caractéristique n'intervient que pour représenter des images tramées, dont nous parlerons dans le chapitre 8 ;

- le type de *machine (device)* sur laquelle travaille l'interpréteur, ainsi que ses possibilités de tramé (l'ordre **currentdevice devicename** empile son nom).

Toutes ces caractéristiques peuvent être modifiées. Cependant, leur manipulation requiert une très bonne expertise en PS ainsi que dans les techniques d'impression. Leur étude déborderait le cadre de cet ouvrage.

7 - SAUVEGARDE-RESTITUTION DE L'ETAT GRAPHIQUE

Il s'agit là d'un des atouts majeurs de PS. Ce langage offre en effet la possibilité, avant de modifier un état graphique, de mettre en réserve (d'empiler) l'état actuel, afin de le récupérer plus tard. Les opérateurs concernés sont

gsave et **grestore**

tous deux sans arguments et n'en déposant aucun sur la pile opérationnelle. Entre ces deux instructions, on travaille avec un état graphique en quelque sorte "privé". Cependant, seules peuvent être qualifiées ainsi les caractéristiques redéfinies ; les autres sont celles de l'état précédent.

> ☞ **Un état graphique donné hérite, pour toutes les caractéristiques non redéfinies, de celles de l'état précédent.**

On rappelle que la sauvegarde d'un état graphique consomme de la mémoire, qui sera libérée par l'instruction **grestore**. Il ne faut donc ni oublier de dépiler les états empilés ni empiler des états dont on ne veut pas retrouver l'usage. De toute façon, PS impose une limite au nombre d'états empilés non dépilés (31 en principe en version 1). On rappelle enfin que ces deux instructions n'affectent pas les autres piles.

Comment se sert-on en pratique de cette facilité ? En général, on crée en début de document un état de référence, avec une police de base, un type de tracé habituel (trait plein), un référentiel placé au coin inférieur gauche de la page, mais avec l'unité fixée à un millimètre. Ces caractéristiques sont en majorité celles établies par défaut par PS. Chaque fois qu'on voudra s'écarter de cet état, pour exécuter une procédure exigeant translation, changement de police ou de type de trait, on écrira

gsave

Une fois cette digression terminée, on revient au système initial par

grestore

Il est relativement exceptionnel d'empiler de nombreux états graphiques.

On peut en donner un exemple très simple. La précédente illustration, page 20, figure 3-7, a été repérée par deux nombres écrits verticalement en petits caractères. L'instruction qui les génère est la suivante :

gsave 120 0 moveto 90 rotate 0.6 .6 scale (3-7) show grestore

L'écriture verticale en petit *corps* n'aura donc été que de courte durée.

♣ Si un opérateur fournit des valeurs numériques de coordonnées, elles sont relatives à l'état graphique en cours. Il ne faudra donc pas, avant de les utiliser, modifier cet état graphique par exemple par **scale**, **rotate**, **translate** ou **grestore**. Cependant, si c'est vraiment nécessaire, on pourra transformer les coordonnées anciennes avec les opérateurs **transform** et **itransform**.

$x_1\ y_1$ **transform** $\rightarrow$ $x_0\ y_0$
$x_0\ y_0$ **itransform** $\rightarrow$ $x_1\ y_1$

Les indices 1 se rapportent au système utilisateur, les indices 0 au système machine.

La remarque s'applique en particulier à **currentpoint** (opérateur décrit page 17).

☞ Notons en terminant que, si toutes les caractéristiques de l'état graphique affectent tous les tracés opérés par **stroke** et les remplissages assurés par **fill** , certaines seulement affectent les *écritures* réalisées par **show** et assimilés. Ce sont la CTM, la police, le pochoir, le chemin et le point courants, ainsi que le niveau de gris, mais ni l'épaisseur, ni la forme du trait, ni son pointillé.

Chapitre 4

PROGRAMMATION STRUCTUREE

Par programmation structurée, on entend un ensemble de moyens permettant de rompre la rigidité d'une exécution purement séquentielle du programme. On parle également de *ruptures de séquence*. PostScript offre dans ce domaine trois possibilités :

- les boucles,
- les procédures,
- les instructions conditionnelles.

Nous allons les étudier tour à tour. Mais il nous faudra auparavant examiner les deux principaux types d'objets composites. On les appelle également *objets structurées*. Ces objets sont en effet tellement liés à la *programmation structurée* que, sans eux, cette dernière perdrait la majeure partie de son intérêt.

Les procédures, quant à elles, sont suffisamment proches des opérateurs pour qu'elles soient étudiées avec eux. Ce sera l'objet du chapitre suivant.

Avant d'examiner les instructions conditionnelles, on va apprendre comment poser une condition en PostScript, ce qui demande tout d'abord d'étudier les expressions logiques.

1 - EXPRESSIONS LOGIQUES

Une expression logique est une expression à valeur booléenne **true** *(vraie)* ou **false** *(fausse)*. Sous sa forme la plus simple, elle comprend un opérateur de comparaison portant sur deux opérandes et s'écrit :

a b op → **true** ou **false** *(sur la pile)*

op =		
	eq *(a* égale *b)*	**ne** *(a* non égal à *b)*
	lt ($a < b$)	**le** ($a \leq b$)
	gt ($a > b$)	**ge** ($a \geq b$)

L'opérateur *op* prélève deux opérandes sur la pile, les compare et laisse le booléen **true** ou **false** , qui servira d'argument à un opérateur ultérieur, en général un **if** . La forme et la signification de ces opérateurs est au premier abord la même qu'en Fortran. Mais ces opérateurs sont très tolérants en ce qui concerne leurs opérandes : sont acceptés les nombres, les caractères, les chaînes, les tableaux ... Avant comparaison de tels objets, les opérateurs tentent en général de les convertir en entiers (les caractères d'une chaîne sont évalués sur la base de leur numéro dans la police en cours).

D'autres opérateurs logiques permettent d'écrire des expressions composées: ce sont **and, or, xor** et **not**, dont la signification est classique. Ils exigent deux opérandes (un seul avec **not**), ne pouvant être que **true** ou **false**. On rappelle brièvement leur algorithme :

a b **and** → **true** seulement si *a* **ET** *b* sont tous deux vrais **(1)**
a b **or** → **true** dès qu'un opérande *a* **OU** *b* est vrai
a **not** → **true** si *a* est faux (et **false** si *a* est vrai)
a b **xor** → **true** si l'**un seulement** des opérandes *a* **OU** *b* est vrai **(2)**

Exemples :

```
10 5 gt                    →  true
1 dup mul 1 ne             →  false
(abc) (abc) eq             →  true
(abc) <616263> eq          →  true
a not b not or a b and not eq →  true
```

```
% 10 est en effet plus grand que 5
% 1 multiplié par 1 est encore égal à 1
% ce sont des chaînes identiques
% idem, voir page 5 (§9)
% quels que soient a et b, car c'est en PS
% l'expression du théorème de De Morgan
```

---

1 - Dans ce livre, le symbole → veut dire : *dépose sur (le sommet de) la pile*.
2 - Attention : les opérateurs logiques **and**, **or**, **xor** et **not** peuvent également agir sur des entiers, mais avec une tout autre signification (cf. page 31).

2 - CHAINES

On sait – et c'est vrai pour tout type d'objet – qu'il existe deux sortes de chaînes, les chaînes constantes et les chaînes variables :

- une **chaîne constante** ou *initiale* est écrite comme une suite de caractères imprimables entre parenthèses (cf. page 5). Les seuls caractères spéciaux tolérés entre ces symboles d'encadrement sont les parenthèses appariées. On rappelle que des séquences d'échappement peuvent y représenter des caractères indisponibles au clavier, dont les spéciaux eux-mêmes. Enfin, une chaîne constante peut s'écrire comme une suite de chiffres *hexadécimaux* encadrés par **<** et **>** . Chaque chiffre donne le numéro du caractère qu'il représente.

- une **chaîne variable** est désignée par un nom, dont l'écriture est soumise à quelques restrictions (cf. p. 4).

Création

La chaîne constante est créée par les parenthèses qui l'encadrent. La chaîne variable doit être créée – ou définie – par l'un des procédés ci-après :

- définition par **string** */ch n* **string def**
- initialisation par **def** */ch* **(** *chaîne constante* **)** **def**

La première instruction définit *ch* comme une chaîne de *n* caractères et la remplit de caractères **null**. La seconde définit *ch* comme une chaîne, recopie en elle le contenu de la chaîne constante placée en argument et lui donne la même longueur que celle-ci. Par exemple,

/automne (Les sanglots longs **\r\n***Des violons)* **def**

définit le nom *automne* comme celui d'une chaîne et lui affecte les vers cités. Sa longueur sera de 32 caractères : 26 lettres, 4 espaces, 2 caractères spéciaux.

Longueur et indice

On peut connaître la longueur d'une chaîne *ch* au moyen de l'instruction

ch **length** → *n* (longueur de la chaîne)

La longueur *n* ou *taille* (nombre entier) est déposée sur la pile. Appliquée à la chaîne *automne* ci-dessus, **length** donnerait 32 (ce n'est pas forcément sa longueur réelle, mais celle fixée lors de sa définition).

Chaque caractère d'une chaîne possède un indice implicite commençant à 0 (caractère de gauche) et se terminant à $n-1$.

Recopie entre chaînes

Il existe trois façons de copier une chaîne dans une autre (chaîne-*source* dans chaîne-*cible)*. La première (**def**), déjà décrite, est sans doute la meilleure : elle opère une copie au sens usuel. La seconde consiste à utiliser **putinterval** (cf. substitution de sous-chaînes). La troisième emploie l'opérateur **copy** , qui agit comme le ferait **putinterval** avec un indice *i* égal à 0 et duplique sur la pile la chaîne-source, ce qui n'est en général pas recherché [3].

/ch2 ch1 **def** ou bien *ch1 ch2* **copy**

La chaîne-source *ch1* est une chaîne constante ou variable, mais la chaîne-cible *ch2* ne peut être qu'une variable. En outre, dans le cas de **copy**, *ch2* doit avoir été définie explicitement comme chaîne variable (par **string def**). Exemples :

/ch1 **(** *abcdef* **)** **def**	*ch1* contiendra exactement *abcdef*
/ch2 50 **string def**	
ch1 ch2 **copy**	*abcdef* est déposé sur la pile, et *ch2* contiendra *abcdef* puis 44 caractères nuls.
/ch3 ch2 **def**	*ch3* devient identique à *ch2* (longueur 50).

Extraction et substitution d'une sous-chaîne

On extrait de la chaîne *ch1* la sous-chaîne *ch2* , grâce à **getinterval**

/ch2 ch1 i m **getinterval def**

L'opérateur **getinterval** extrait de *ch1* une sous-chaîne de *m* caractères commençant au *i*ème et la dépose sur la pile ; puis **def** la retire de la pile et la place dans *ch2* .

Inversement, **putinterval** substitue une sous-chaîne *ch2* dans *ch1* :

ch1 i ch2 **putinterval**

Toute la chaîne *ch2* (longueur l_2) remplace dans *ch1* une sous-chaîne de même longueur débutant au *i*ème caractère. Les longueurs respectives l_1 et l_2 de *ch1* et *ch2* doivent être compatibles. Il faut que $l_1 \geq (l_2 + i)$. En outre, *i* ne doit pas être négatif. L'opérateur **putinterval** ne dépose rien sur la pile.

Exemples :

automne 1 3 **getinterval** → **(es)** % *chaîne "automne" définie p.* 39
(*automne***)** 1 3 **getinterval** → **(uto)** % *chaîne constante nouvelle*
automne 5 **(** *tridulations* **)** **putinterval** → % *rien sur la pile*, mais :
automne → **(Les stridulations \r\nDes violons)**

3 - Est déposée sur la pile la partie originelle de *ch2* qui sera modifiée par **copy**.

Recherche de sous-chaînes

On recherche la première apparition de la sous-chaîne *sch* dans la chaîne *ch* au moyen de **search** :

ch sch **search** → *ch2 sch ch1* **true** (si *sch* fait partie de *ch*)
→ *ch* **false** (si *sch* n'est pas dans *ch*)

(*ch1* et *ch2* désignent les fractions de *ch* précédant et suivant *sch*).

On cherche si la sous-chaîne *sch* est placée au début de la chaîne *ch* par :

ch sch **anchorsearch** → *ch2 sch* **true** (si *sch* est le début de *ch*)
→ *ch* **false** (dans le cas contraire)

Exemples :

(*abcdefgh* **) (** *cd* **) search** → **(** *efgh* **) (** *cd* **) (** *ab* **) true**
(*abcdefgh* **) (** *pq* **) search** → **(** *abcdefgh* **) false**
(*abcdefgh* **) (** *abc* **) anchorsearch** → **(** *defgh* **) (** *abc* **) true**
(**search** s'emploie souvent de manière répétitive, cf. page 28).

Manipulations de caractères

On récupère sur la pile le numéro du *i*ème caractère de la chaîne *ch* grâce à :

ch i **get** → numéro du *i*ème caractère de *ch*

et on **substitue** le caractère de numéro *ncar* au *i*ème caractère de *ch*, avec :

ch i ncar **put**

(i doit être inférieur à la longueur de *ch*, il commence à 0. Pas de dépôt sur la pile).

Exemples : **(** *abcdefgh***)** 2 **get** → 99 % *(valeur numérique décimale de c)*
/ch **(***valeur=90 F***) def** *ch* 8 57 **put** → ∅ % *ch* devient *(valeur=99 F, n° ASCII de* '9' *= 57)*

Les opérateurs **put** et **get** retirent la chaîne *ch* de la pile. Ils sont surtout utiles dans des boucles, tandis que **putinterval** et **getinterval** sont en quelque sorte des boucles implicites.

> ☞ **A propos des chaînes et des tableaux**
>
> Respecter longueurs et indices, sinon on obtiendra l'erreur **rangecheck**.
>
> Se rappeler que l'indice commence à 0 et se termine à *l*–1.
>
> En cas de message d'erreur signalant des objets **null** [4], vérifier si on ne tente pas d'employer une chaîne créée par **string**, mais non initialisée ou initialisée avec une chaîne de longueur insuffisante.
>
> Enfin, se souvenir qu'en PS il n'y a pas de chaînes élastiques comme en Basic. Leur longueur est définitivement fixée à leur création.

3 - TABLEAUX

Les tableaux sont des objets composites voisins des chaînes, mais plus généraux. Ils peuvent contenir des éléments quelconques : nombres, chaînes, opérateurs, procédures ou *tableaux*. PS repère ces éléments par un indice variant de 0 à *n*–1, *n* étant la taille du tableau. Les tableaux se manipulent comme les chaînes et avec presque les mêmes opérateurs. Exceptions :

- **string** est remplacé par **array** dans l'instruction créant un tableau vide,
- deux nouveaux opérateurs sont disponibles : **astore** et **aload** .

La liste des opérateurs disponibles est résumée page suivante.

4 - La valeur **null** en PS correspond au caractère ASCII numéro 0, c'est-à-dire à celui marquant les fins de chaîne en C ou en langage-machine.

/tab1 n **array def**	crée le tableau *tab1* de *n* éléments, tous **null** (5).
/tab2 **[** a_0 a_1 **...** a_{n-1} **] def**	crée le tableau *tab2*, lui affecte les *n* éléments du tableau **[**a_0 ... a_{n-1}**]** (6) et lui donne la taille *n*.
b_0 b_1 **...** b_{n-1} *tab* **astore**	remplit *tab*, déjà défini et de taille *n*, avec les *n* éléments b_0 ... b_{n-1} (le nombre d'éléments doit être égal à la taille du tableau) (6).
/tab3 c_0 c_1 **...** c_{n-1} *n* **array astore def**	crée le tableau *tab3* de taille *n* et le remplit avec les *n* objets c_i.
tab **aload**	empile tous les éléments de *tab*, puis *tab* (7).
tab **length**	empile *n* (taille du tableau *tab).*
tab1 tab2 **copy**	copie dans *tab2* le tableau contenu dans *tab1* (8).
tab i m **getinterval**	empile un sous-tableau dérivé de *tab*, comportant *m* éléments à partir du *i*ème.
tab1 i tab2 **putinterval**	remplace les éléments de *tab1*, à partir du *i*ème, par les éléments de *tab2*.
tab i **get**	empile le *i*ème élément du tableau *tab*.
tab i objet **put**	remplace par *objet* le *i*ème élément de *tab*.

Exemples

	Effet sur la pile
... */agenda* 7 **array def**	→ **...**
... */drapo* [*(bleu) (blanc) (rouge)*] **def**	→ **...**
... */mois* [31 28 31 30 31 30 31 31 30 31 30 31] **def**	→ **...**
1 2 3 4 5 6 7 *agenda* **astore**	→ *agenda*
agenda 1 3 **getinterval**	→ [2 3 4]
drapo **aload**	→ *(bleu) (blanc) (rouge)* [*(bleu)(blanc)(rouge)*]
agenda **length**	→ 7
... drapo 1 3.1416 **put**	→ ...
... agenda 3 *drapo* **putinterval**	→ ...
agenda	→ [1 2 3 *(bleu)* 3.1416 *(rouge)* 7]
drapo 1 **get**	→ 3.1416

♣ *ncar ch i* 3 2 **roll put** place le caractère de numéro *ncar* dans *ch* en position *i* (très utilisé).

4 - BOUCLES

Outre **pathforall**, étudiée page 39, PostScript offre quatre possibilités de boucles, dont les opérateurs sont **loop**, **repeat**, **for** et **forall**. Ils ont pour argument une procédure qui peut être désignée soit par son nom, soit par une liste d'instructions. Dans tous les cas, cette procédure doit être placée entre accolades.

- loop { *procédure* } loop

La procédure-argument est exécutée indéfiniment, à moins qu'elle ne comporte l'opérateur **exit** , auquel cas on quitte la boucle en cause (l'exécution reprend à la première instruction située après ladite boucle). Le plus souvent, l'opérateur **exit** dépend d'un **if** .

- repeat *n* { *procédure* } repeat

Comme indiqué, la procédure est exécutée *n* fois.

- for n_1 *i* n_2 { *procédure* } for

La procédure est exécutée $1 + (n_2 - n_1)/i$ fois. Elle crée une variable, qu'on appelle *indice ou variable de boucle*, lui donne d'abord la valeur n_1, l'augmente à chaque exécution de la valeur *i* (incrément) et s'arrête lorsque l'indice dépasse la valeur n_2 (si *i* est positif, sinon, quand *i* devient inférieur à n_2). De plus, **for** place sur la pile, avant chacune des exécutions, la valeur de cet indice, qu'on pourra ainsi utiliser, par exemple pour indicer l'élément d'une chaîne ou d'un tableau. Si on ne l'utilise pas, il faudra penser à le détruire (avec **pop**), sinon son accumulation sur la pile deviendrait vite gênante.

Si $i>0$ et $n_1>n_2$ ou si $i<0$ et $n_1<n_2$, la procédure n'est pas exécutée. *i*, n_1, n_2 sont réels ou entiers.

---

5 - **array** dépose sur la pile le tableau-opérande *tab1* (mais pas ses éléments ! C'est un pointeur.)
6 - Aucun autre séparateur que l'espace ne doit délimiter les éléments d'un tableau.
7 - Attention : empile *tab* , en plus de ses éléments. Si nécessaire, le retirer avec **pop**.
8 - Dépose sur la pile la partie originelle de *tab2* , celle qui sera modifiée par **copy**.

- forall *objet-composite* **{** *procédure* **}** **forall**

Deux arguments sont nécessaires à l'opérateur **forall**, une procédure bien sûr, comme dans les autres types de boucle, et un objet composite tel que chaîne, tableau, dictionnaire...

Avant chaque exécution, **forall** place sur la pile un élément du tableau (ou de la chaîne) en commençant par celui d'indice 0. Dans le cas d'une chaîne, c'est bien sûr le numéro du caractère qui est empilé. Dans le cas d'un dictionnaire, c'est le couple nom-valeur qui l'est. Il y a autant d'exécutions que d'éléments dans l'objet composite. Si la procédure n'utilise pas les éléments empilés ou s'ils ne sont pas détruits par **pop** , ils s'accumuleront sur la pile.

☞ A propos de l'indice : avec **for** , un indice de boucle est automatiquement créé et géré. Si on en a besoin avec un autre opérateur, on peut créer une variable numérique, l'initialiser et l'incrémenter à chaque itération. Exemples :

/*i* 0 **def**
chaîne **{***procédure ...* **/***i i* 1 **add def}** **forall**

0 *chaîne* **{***procédure-caractère* **dup** (un seul dépilage de l'indice) 1 **add}** **forall** **pop**

☞ Toutes ces boucles peuvent être imbriquées. La notation polonaise ne facilitant pas le repérage des boucles multiples, il est prudent de *commenter*.

Exemples de boucles

On a déjà employé des boucles dans nos exemples, comme dans la répétition de courbes avec **lignarrond** ou avec la ligne brisée page 12. Autres exemples :

... **arcto 4 { pop } repeat** (supprime les 4 valeurs laissées sur la pile par **arcto**)

1 2 3 4 3 **{ add } repeat** → 10

2 –.5 -2.4 **{ } for** → 2 1.5 1 0.5 0 –0.5 –1 –1.5 -2 (indices sur la pile)

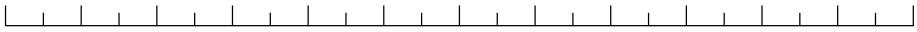
0 1 3 **{ (***abcdefgh***) exch get } for** → 100 88 98 97

0 [31 28 31 30 31 30 31 31 30 31 30 31] **{ add } forall** → 365

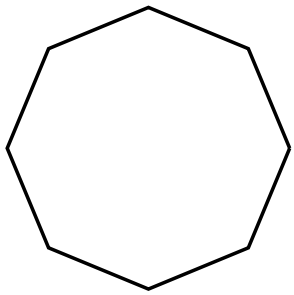
```
/tabxy 5 5 16 17 30 6 37 12 51 3 10 array astore def /i 0 def
tabxy aload pop 5 { i 0 eq { moveto } { lineto } ifelse /i 1 def} repeat stroke
/g 1 def /d 2.5 def tabxy aload pop { point } repeat % effacement par gros points blancs
/g 0 def /d 0.7 def tabxy aload pop { point } repeat % points noirs, rayon 0.7 mm.
```

Les quatre lignes ci-dessus tracent la ligne brisée en haut de la page 12 à droite, pour des points dont les coordonnées ont été rangées dans le tableau *tabxy* et en utilisant la procédure *point* définie à la page 11.

```
0 0 moveto 0 10 110 {0 2.2 rlineto 0 moveto 5 0 rlineto currentpoint 0 1.4 rlineto moveto 5 0 rlineto} for
0 2.2 rlineto stroke % donne la ligne ci-dessous
```

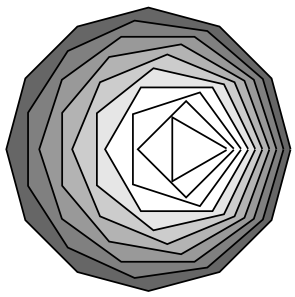


La boucle ci-après dessine un polygone régulier de *n* côtés (*n* quelconque), de rayon *a* et centré en *x, y* :



```
/a 120 def /n 8 def /r 360 n div def /x 140 def /y 450 def
x y translate 3 setlinewidth a 0 moveto % point de départ
n {r rotate a 0 lineto} repeat stroke % boucle, n côtés
```

et voici la même boucle englobée dans une autre qui dessine 10 polygones l'un au-dessus de l'autre, légèrement décalés avec *n* variant de 12 à 3, dégressifs en rayon et en grisé :



```
/a 120 def 0 150 translate /g .3 def
12 –1 3 { /n exch def /r 360 n div def
  /g g .1 add def gsave x 0 translate a 0 moveto
  n {r rotate a 0 lineto} repeat
  gsave stroke grestore g setgray fill
  /a a 10 sub def /x x 4 add def grestore
} for
```

5 - EXECUTION CONDITIONNELLE

En PS, l'exécution d'instructions peut dépendre de la valeur d'une condition, cette valeur ne pouvant être que celle des booléens **true** ou **false.** Il est même possible d'aiguiller l'exécution sur un bloc d'instructions ou sur un autre selon la valeur de cette condition. On écrit alors l'une des deux formes :

booléen **{** *bloc d'instructions ou procédure* **}** **if**
booléen **{** *bloc n° 1* **}** **{** *bloc n°2* **}** **ifelse**

La signification en est évidente :

- avec **if** , si le booléen est **true**, le *bloc* est exécuté ; si le booléen est **false** , il n'y a pas exécution ;

- avec **ifelse**, l'un des deux blocs est toujours exécuté. Si le booléen est **true**, c'est le *bloc n° 1*. Si c'est **false**, c'est le deuxième.

Remarquons que le *bloc* peut être remplacé par une (ou plusieurs) procédure. Il n'y a aucune différence en PS entre *bloc* et *procédure*. Dans tous les cas, blocs ou procédures sont entourés d'accolades.

Bien sûr, le lecteur aura compris que, dans la majorité des cas, les booléens **true** ou **false** sont déposés sur la pile par l'une des expressions logiques exposées au début du chapitre (page 23) ou par des opérateurs comme **search**, **anchorsearch**, ... ainsi que (on le verra dans le chapitre 6) **read**, **readline**, **readstring**, **known** ...

Exemples :

1 - Vérifie que *y* est positif avant passage au logarithme :

```
/logy y dup 0 gt { log } {pop -1E38 } ifelse def      % y>0 ? | oui → logy = log(y),
                                                      % non → logy = –10^38.
```

2 - Trace le graphe de *n* couples de valeurs x_i, y_i ($0 \le i < n$) contenues dans *tab*. Initialise le chemin avec **moveto** appliqué au premier point, puis exécute des **lineto** sur les points suivants.

```
tab aload pop 0 1 n { 0 eq { moveto } { lineto } ifelse } for stroke
```

3 - Recherche le plus grand nombre d'un tableau de nombres positifs *tab*

```
/xmax 0 def  tab {dup xmax gt {/xmax exch def } { pop } ifelse} forall
```

4 - Solution de l'équation du 2e degré, $ax^2 + bx + c = 0$ avec $delta = b^2 - 4ac$:
(comporte deux **if** imbriqués)

```
delta 0 gt  { /x1 b delta sqrt sub  -2 a mul div def  /x2 b a div x1 add neg def }
            { delta 0 eq  {/x1 b neg a div  2 div def /x2 x1 def }
                          { /x1 null def /x2 null def }  ifelse
            } ifelse
```

5 - Cet exemple, avec un **search** répétitif, recherche combien il y a de lettres *e* dans une phrase :
(combien y-a-t-il de lettres e dans cette petite phrase ?)

```
/n 0 def {(e) search {pop pop /n n 1 add def} {n == exit} ifelse } loop
```

L'interpréteur écrira ou affichera 10. Pour comprendre cette instruction, ne pas oublier que **search** dépose 4 objets sur la pile, le dernier étant un booléen qui aiguille le **ifelse** (on sort de la boucle grâce à **exit** quand il n'y a plus de sous-chaîne *e*). S'il y en a, on détruit deux objets-chaîne inutiles par les **pop**, on garde la partie de la chaîne qui suit le *e* trouvé et on recommence (à cause du **loop**) sur ce tronçon de chaîne.

Chapitre 5

OPERATEURS ET PROCEDURES

Nous avons parfois utilisé des notions relativement simples comme celles d'instruction, d'opérateur, d'opérande, sans en avoir précisé les contours. Nous allons maintenant un peu mieux définir ces notions, pour asseoir nos connaissances et ouvrir des horizons nouveaux vers le traitement interne des objets de PS.

1 - INSTRUCTION ELEMENTAIRE

L'instruction PS comporte toujours au moins un opérateur et très souvent des opérandes (ou arguments). Les opérandes sont écrits **avant** les opérateurs et l'interpréteur les empile au fur et à mesure de leur arrivée. Quand survient un opérateur, PS le reconnaît (parce qu'il trouve sa définition dans le dictionnaire **systemdict**) et l'exécute. Cet opérateur peut exiger la présence d'un chemin (comme **stroke**, **fill** ...), d'un point courant (comme **lineto**, **curveto**, **arcto** ...) et–ou d'opérandes qu'il prélèvera sur la pile.

Si la pile ne comporte pas assez d'opérandes pour l'opérateur en cours d'exécution, il y aura erreur par atteinte du bas de pile (**stackunderflow**).

On a vu également que certains opérateurs exigent comme opérandes des procédures (**for , forall , repeat , loop , if , ifelse ...**) et que **if** et **ifelse** exigent en plus la présence sur la pile de l'un des booléens **true** ou **false**, fourni par les opérateurs de comparaison tels que **gt**, **lt**, **le**, **ge**, **eq**, **ne**, éventuellement associés entre eux grâce à **and**, **or**, **xor** ou **not**.

2 - OPERATEURS MATHEMATIQUES

Nous avons déjà souvent rencontré dans nos exemples des opérateurs mathématiques élémentaires. Chacun de ces opérateurs prélève sur la pile un ou deux opérandes, symbolisés ci-après par *a* et *b* , ou *i* et *j* si ce sont strictement des entiers, puis il place sur la pile le résultat – unique – de son opération :

add : *a b* **add** → *a*+*b* *
mul : *a b* **mul** → *ab* *
neg : *a* **neg** → – *a* *

sub : **a b sub** → **a** – **b** *
div : **a b div** → **a**/**b** *(réel)*

On pourra également employer des opérateurs un peu moins courants :

idiv : *i j* **idiv** → entier, partie entière de *i*/*j* (Ex.: 3 2 **idiv** → 1, mais –3 2 **idiv** → –1)
mod : *i j* **mod** → entier, *i* modulo *j* (reste de la division ci-dessus)
abs : *a* **abs** → valeur absolue de *a* (= *a* si *a*≥0, –*a* si *a*<0) *
ceiling : *a* **ceiling** → entier juste supérieur à *a* (Ex.: 2.5 **ceiling** → 3, mais –2.5 **ceiling** → –2) *
floor : *a* **floor** → entier juste inférieur à *a* (Ex.: 2.5 **floor** → 2, mais –2.5 **floor** → –3) *
round : *a* **round** → entier le plus proche de *a* (Ex.: 2.5 **round** → 3, mais –2.5 **round** → –2) *
truncate : *a* **truncate** → partie entière de *a* (–2.9 **truncate** → –2) *

ainsi que les fonctions :

sqrt : *a* **sqrt** → réel, racine carrée de *a* , (*a*≥0)
sin : α **sin** → sin α (α est en degrés)
cos : α **cos** → cos α (id)
atan : *a b* **atan** → α en degrés, égal à l'arctangente de *a*/*b*, *a* et *b* ne doivent pas être tous deux nuls)
exp : *a b* **exp** → *a* puissance *b* (si *a*<0, *b* doit être entier)
ln : *a* **ln** → Log *a*, logarithme népérien de *a* (*a*>0)
log : *a* **log** → log *a*, logarithme décimal de *a* (*a*>0)
rand : **rand** → entier au hasard, entre 0 et $2^{31}-1$
srand : *a* **srand** → initialisation de la fonction **rand** (donne une séquence reproductible)

---

* le type du résultat est entier, si les deux opérandes sont entiers, sinon il est réel.

Quelques remarques

1 - Les fonctions trigonométriques ne sont donc pas toutes disponibles.

2 - **atan** fournit l'angle exact – dans le bon quadrant – grâce à ses deux opérandes. Cet angle, jamais négatif, est compris entre 0 et 360°.

3 - Quelques limites sont imposées aux opérandes, mais elles sont classiques.

4 - **srand** est utile pour retrouver avec **rand** la même séquence de nombres aléatoires au cours de différente sessions.

3 - OPERATEURS DE CONVERSION

Des conversions entre *types* d'objet sont souvent nécessaires, spécialement entre nombres et caractères. Chacun d'eux, comme d'habitude prélève sur la pile son ou ses opérandes et y dépose son résultat. Voici les **principaux** opérateurs de conversion de PostScript :

cvs :	*n ch* **cvs**	→	place dans la chaîne *ch* la représentation du nombre *n* (et la copie sur pile)
cvrs :	*n b ch* **cvrs**	→	place dans *ch* la chaîne représentant le nombre *n* en base *b* (id)
cvi :	*n* **cvi**	→	empile un entier représentant le nombre *n*, éventuellement tronqué (id)
	ch **cvi**	→	convertit la chaîne *ch* en nombre entier et empile le résultat (id)
cvr :	*n* ou *ch* **cvr**	→	convertit l'entier *n* ou la chaîne *ch* en réel et empile le résultat (id)
cvx :	*nom* **cvx**	→	convertit *nom* en l'objet exécutable défini par *nom* (parfois nécessaire avant **exec**)
load :	/*nom* **load**	→	met sur la pile le contenu de la variable *nom*, convertit un nom littéral en son contenu **(1)**.

Les opérateurs de conversion nombre-chaîne sont nécessaires pour imprimer des résultats dépendant de calculs, lesquels fournissent des valeurs numériques, alors que l'opérateur d'impression – **show** – exige une chaîne comme argument. Dans les autres langages, c'est la fonction d'écriture (**print**, **write**) qui, à l'aide du format, convertit le nombre en chaîne, seule forme imprimable. Quelques exemples :

/*ch* 4 **string def** 1992 *ch* **cvs** → **(**1992**)** , chaîne imprimable
1992 20 **add ( ) cvs show** % imprime 2012
/*ch* 16 **string def** -1 2 *ch* **cvrs** → (1111111111)

4 - AFFECTATION

Dans beaucoup de langages, on *affecte* une valeur à une variable grâce au signe = (ou au signe := en Pascal et en Ada, pour bien marquer qu'il ne s'agit pas d'une *égalité*) **(2)**. En PS, on affecte une valeur à une variable grâce à **def** ou **store** **(3)** , comme on a affecté une procédure à une variable avec **def**.

/*i* 0 **def** /*i i* 5 **add store**
/*i* **dup load** 5 **add store**

La première instruction crée la variable *i* et la met à 0. Les autres ajoutent 5 à la valeur de *i* . C'est moins simple qu'en Basic (on écrirait *i*=*i*+5) et surtout qu'en C (*i* += 5). Dans ces instructions, **def** et **store** sont apparemment permutables. Si la variable citée existe déjà, ils lui affectent la valeur opérande. Sinon, ils créent une nouvelle entrée dans le dictionnaire **userdict** , et lui donnent la valeur de l'opérande.

5 - CHARGEMENT

En informatique *classique*, on charge des registres. En PS, on ne peut charger que la pile. C'est ce que fait l'interpréteur quand il rencontre une constante ou une variable (dans ce dernier cas, il aura cherché auparavant sa valeur dans un dictionnaire).

Il faut bien distinguer **adresse** de variable – qui s'écrit /*var* – et **contenu**, qui s'écrit *var* tout simplement. On peut passer de l'adresse au contenu précisément par l'opérateur **load**, qui charge le contenu sur la pile, tandis que /*var* permet l'affectation de la variable.

/*var* **dup load** ... *instructions* ... **def** (PS)
⇔ *var = var opérateur expression* ... (classique)

Malgré tout, l'emploi de **load** est peu courant. Le chargement usuel se fait en citant le nom de la variable.

---

1 - **load** n'est pas à proprement parler un opérateur de conversion, mais c'est probablement dans cette section qu'un utilisateur en difficulté chercherait à trouver un tel opérateur.

2 - Est-il bien logique d'écrire *toto* = *toto* + 9 ?

3 - L'opérateur = existe bien en PS, mais il ne sert ni à comparer ni à affecter (cf. en bas de la page 32).

Pour charger sur la pile le contenu de variables composites, il faut utiliser l'opérateur **aload**. Attention, les éléments sont placés de sorte que le plus haut – le premier disponible – soit celui d'indice le plus élevé.

tableau **aload** → a_0 a_1 a_2 ... a_{n-1}

6 - OPERATEURS DE NIVEAU BINAIRE

Quelques opérateurs permettent de faire des opérations au niveau du bit :

and : *m n* **and** → intersection entre les bits des **entiers** *n* et *m* (filtrage)

or : *m n* **or** → réunion entre les bits des **entiers** *n* et *m*

xor : *m n* **xor** → réunion exclusive entre les bits des **entiers** *n* et *m*

not : *n* **not** → opposé bit à bit de l'entier *n* , donc *–(n*+1)

bitshift : *a n* **bitshift** → décale la représentation binaire de l'entier *a* de *n* positions vers la gauche *(n*>0) ou la droite *(n*<0) **(4)**.

Ces opérateurs laissent tous un entier sur la pile. Pour les utiliser, il faut connaître la façon de représenter les entiers (et celle des négatifs par complément à 2 pour **not**). Si les arguments *m* et *n* ne sont pas des entiers, l'erreur **typecheck** est générée ; les convertir auparavant par **cvi** **(5)**.

7 - PROCEDURES

Comme on l'a déjà dit, une procédure est un bloc d'instructions entouré d'accolades. Par exemple, dans les boucles ou avant un **if** , l'un des opérandes doit être une procédure.

On peut également définir des procédures pour éviter d'écrire plusieurs fois la même séquence d'instructions. C'est un procédé extrêmement employé. La procédure *nomproc*, une fois définie par

/*nomproc* **{**séquence quelconque d'instructions**} def**

est utilisée en citant simplement son nom (sans accolades, sauf si elle sert d'opérande dans une boucle ou devant un **if** ou un **ifelse**). PS étant interprété, il est indispensable que **la procédure ait été définie avant tout emploi**.

La procédure prélève sur la pile les arguments nécessaires à ses opérateurs, si nécessaire. Comparé aux autres langages de programmation, PS ne possède que des **variables communes** ou **globales**. Les procédures PS ressemblent plutôt à des macro-instructions ou bien à des sous-programmes GWBasic.

Une procédure n'est pas du tout autonome par rapport au programme principal, terme que d'ailleurs ignore PS. C'est simplement un moyen de condenser l'écriture et d'éviter des redites. On peut admettre qu'une procédure est en quelque sorte *encapsulée* dans un paquet portant son nom, mais à son appel, tout son contenu sera *déballé* sur la pile et les opérateurs de la procédure devront exploiter la pile comme si tout avait été directement écrit. Même le couple **gsave ... grestore** ne peut pas rendre *locales* des variables (elles ne font pas partie de l'état graphique). La seule possibilité de créer de telles variables consiste à définir et utiliser un **dictionnaire temporaire** (voir page 34).

La procédure est spécialement employée pour créer des objets graphiques *a priori* avec par exemple leur point de référence en (0,0) et des dimensions arbitraires. On placera ensuite cet objet dans la page en *(x,y)* et en lui donnant les dimensions *(a,b)* grâce aux opérateurs **translate** et **scale**, comme on l'a déjà indiqué page 13 au sujet des procédures *rect* et *rectar*.

Par exemple, nous avons commencé l'écriture de la procédure *rectar* en ces termes :

/*rectar* **{ /***r* **exch def /***b* **exch def**
/*a* **exch def gsave translate .... } def**

on la fera ensuite exécuter ainsi

x' y' a' b' r' **rectar**

Si l'on se souvient du rôle de l'opérateur d'affectation **def** *(/nom valeur* **def**), il y aura bien affectation de *r'* à *r*, puis de *b'* à *b* et enfin de *a'* à *a* dès le début d'exécution de la procédure. Mais, à moins d'être réaffectées, ces variables sont définies pour tout le reste du travail. Leur place en mémoire et dans les dictionnaires est définitivement réservée.

Toujours dans la procédure *rectar*, noter l'analogie entre le rôle de *x'* et *y'* et celui de *a', b', r'* au moment de l'appel, alors que ces valeurs sont exploitées apparemment de manière différente dans la procédure: *x'* et *y'* deviennent en effet opérandes pour **translate**.

---

4 - Les bits débordants sont perdus, les nouveaux mis à 0.

5 - Les 4 premiers opérateurs sont les mêmes que ceux agissant sur les booléens, cf. p.38. Au point de vue logique, on constate que **true** et **false** sont équivalents aux entiers 1 et 0.

La **récursivité** est une particularité intéressante des procédures PS : cela veut dire qu'elles peuvent s'appeler elles-mêmes. L'exemple classique donné à ce sujet est celui du calcul de la fonction factorielle. En matière de dessin, la récursivité peut être utile, par exemple à coup sûr si l'on se propose de représenter des fractales (figures géométriques autosimilaires).

☞ **Rappel** : le langage PS étant interprété, une procédure doit toujours avoir été définie avant son emploi.

Exemples :

- procédures souvent créées pour simplifier des noms d'opérateur :

/*M* **{moveto} def** **/***pc* **{currentpoint} def** **/***F* **{closepath} def**

- procédures résumant des séquences d'opérateurs usuelles :

/*act* **{ arcto 4 {pop} repeat } def** **/***S* **{ moveto show} def**

- procédure *boul* dessinant des boules de couleur uniforme (page 15 à gauche) :

/*boul* **{***r* **add moveto** *pc r* **sub** *r* 90 450 **arc gsave setgray fill grestore stroke } def**
/*r* 12 **def** .5 12 0 .5 0 12 .5 -12 0 .5 0 -12 4 **{** *boul* **} repeat**
.9 6 6 .9 -6 6 .9 -6 -6 .9 6 -6 4 **{** *boul* **} repeat** .97 0 0 *boul*

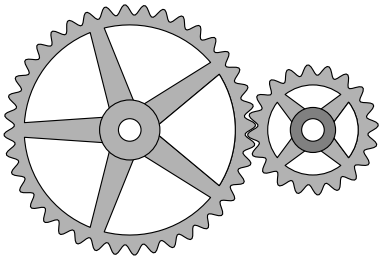
(les nombres .5, .9 et .97 sont les opérandes de **setgray** , les autres sont opérandes de **moveto**.
L'origine de ces coordonnées est au centre de la figure).

Autre exemple produisant le dessin des engrenages ci-après (l'unité est le mm) :

/*gris* **{ gsave setgray fill grestore } def** **%** *argument = taux de gris*
/*sector* **{** *r1* **0 moveto 0 0** *r2 as* **neg** *as* **arc closepath** 1 *gris* **stroke } def**
/*trou* **{** *r0* **0 moveto 0 0** *r0* **0 360 arc** *gris* **stroke } def**
/*couronne* **{** *r3* **0 moveto 0 360** *nd* **div 10 div 360 {***dent***} for closepath 0.7** *gris* **stroke } def**
/*dent* **{** *lang* **exch def** *ang nd* **mul sin** *h* **mul** *r3* **add dup** *ang* **cos mul exch** *ang* **sin mul lineto } def**

% grande roue, *nd*=40 dents, hauteur *h*, 5 secteurs d'angle *as* (axe en 0,0)

/*r0* **4 def** **/***r1* **3 def** **/***r2* **14 def** **/***r3* **16 def** **/***h* **.7 def** **/***as* **32 def** **/***nd* **40 def** % couronne



0 72 360 {gsave rotate *sector* **grestore} for**
.7 *trou* **/***r0* **1.5 def** **1** *trou*

% petite roue, 20 dents, 4 secteurs

24.3 0 **translate** **/***r1* 2 **def** **/***r0* 3 **def**
/*r2* 6 **def** **/***r3* 8 **def**

/*nd* 20 **def** *couronne* **/***as* 38 **def**
0 90 360 **{gsave rotate** *sector* **grestore} for**
0.7 *trou* **/***r0* 1.5 **def** 1 *trou*

8 - OPERATEURS D'EXECUTION

Quelques opérateurs sont à connaître, quoique d'un emploi plus rare. On a déjà rencontré

exit

qui sert à sortir d'une boucle, surtout de celles de type **loop** . On rappelle qu'en cas de boucles imbriquées on passe alors à celle de niveau immédiatement supérieur (ou juste plus externe).

On peut quitter une session PostScript par **quit .**

Les opérateurs **bind def** permettent de diminuer le temps d'exécution des procédures en remplaçant dans le dictionnaire leur définition par une expression de bas niveau (c'est une sorte de compilation).

L'opérateur **exec** est parfois nécessaire ; il fait exécuter la procédure dont le nom est en haut de pile (ce qui exige souvent une conversion par **cvx**). Il peut faire exécuter des procédures en boucle, ou bien en choisir une dans un tableau.

{ *proc1* **} {** *proc2* **} {** *proc3* **}** 0 2 5 **{ ... exec ... } for**
[{ *proc1* **} {** *proc2* **}** ...**]** **/***i* 2 **def** *i* **get cvx exec**

Nous avons mentionné, page 9, le rôle essentiel de **showpage** .

Citons enfin des opérateurs qui n'ont d'intérêt qu'en programmation interactive : ils provoquent une écriture dans le fichier de sortie.

= ou **==**	retire l'objet du haut de la pile et envoie sa valeur sur la ligne de sortie.
stack ou **pstack**	envoient sur la ligne de sortie la représentation de la pile toute entière sans la détruire.
flush	vide le tampon de sortie. Envoie sur la ligne de sortie le contenu éventuel de ces tampons.

On comprendra mieux l'intérêt de ces opérateurs dans le chapitre 9.

Annexe 8

GHOSTSCRIPT et GSVIEW

Ghostscript (GS) est un interpréteur PostScript de haute qualité élaboré par *Aladdin Enterprises* et spécialement par L. Peter Deutsch, son président. Ce logiciel est distribué gratuitement sur *Internet* par l'association *Free Software Foundation* (distribution GNU ou Aladdin, serveurs ftp.cs.wisc.edu, ftp.inria.fr, ftp.jussieu.fr...). Aladdin Enterprises fait sienne la position de la *League for Programming Freedom*, prônant le logiciel libre ou, tout au moins, bon marché.

Le télé-chargement de GS se fait sous *ftp anonymous*. Si on n'a pas accès à Internet, on s'adressera à un ami (la copie est autorisée) ou à Russel Lang (Australie) qui envoie à tout demandeur un disque (optique) contenant GhostScript, moyennant le prépaiement des frais d'expédition. GS peut fonctionner sous Unix, DOS, DOS-Windows, Macintosh, ...

Le logiciel reçu est comprimé (par **pkzip** en principe) ; après décompression (par **gunzip** dans le cas présent), GhostScript comprendra **(1)** :

- des fichiers d'informations (README, USE.doc, MAN.doc, ...), ainsi que le texte de la licence d'exploitation (fichier général COPYING ; fichier COMMPROD.doc en cas d'usage commercial),

- des programmes exécutables permettant d'interpréter à l'écran les fichiers PostScript et leurs fichiers prologues (initialisations),

- des pilotes *(drivers)* permettant d'imprimer l'interprétation des fichiers PS sur de nombreuses imprimantes laser, à jet d'encre, à bulles ou à aiguilles,

- des jeux de polices de caractères, conformes à PS, mais non identiques à celles d'Adobe (si on le désire, on peut remplacer ces polices par celles d'Adobe, à condition de respecter les droits de cette firme).

Le logiciel occupera 5 à 7 méga-octets de disque (en version 5). Il n'est pas nécessaire d'installer les fichiers sources (***.C**, ***.H**, ***src.zip**). Avec les versions 4 ou 5, trois fichiers comprimés de moins de 1,4 Mo chacun sont nécessaires : **gs***nnn***ini.zip**, **gs***nnnsys***.zip** et **gs***nnn***fn1.zip** (remplacer *nnn* par le symbole de la version et *sys* par celui du système, **w32** sous Windows 95 par exemple).

Deux fichiers de polices sont disponibles : ***fn1.zip** (ou ***fonts-std***) contient les polices courantes (y compris **symbol** et **dingbats**) ; ***fn2.zip** (ou ***fonts-other***), facultatif, contient des polices plus raffinées (gothiques, cursives, ...). Les fichiers produits par décompression de chacun de ces deux jeux doivent être placés dans un même répertoire. On ne modifiera pas le fichier **Fontmap**, mais on doit affecter à la *variable d'environnement* **GS_LIB** les noms des répertoires contenant GS et ses polices (voir page suivante).

GS ouvre deux fenêtres : l'une affiche erreurs, avertissements et réponses aux ordres **==**, **pstack** ..., l'autre contient le dessin résultant du fichier. La vitesse d'exécution de GS et sa précision sont remarquables. Dessins et images apparaissent à l'écran bien plus vite que sur une imprimante. GS reconnaît automatiquement la carte graphique et fonctionne bien avec EGA, VGA et diverses Super-VGA (liste dans **DEVS.mak**). Il détecte bien les erreurs, mais paraît plus tolérant que beaucoup d'imprimantes. Il est facile de faire interpréter à GS avant le fichier principal un fichier prologue (écrire un fichier ***.bat** ou donner successivement le nom des deux fichiers ; c'est moins simple avec Gsview). Seules ombres au tableau, les opérateurs **image** et, en niveau 2, **colorimage** ne semblent pas exploiter toute la dynamique couleur des cartes SVGA, du moins jusqu'à la version 5.01 ; de même, les hautes résolutions des imprimantes ne sont pas toujours prises en compte.

Son aptitude à imprimer des fichiers PS sur des imprimantes ordinaires est du plus haut intérêt. Pour ce faire, il suffit par exemple (mais d'autres moyens sont disponibles) d'ajouter dans le programme la ligne

(*nom* **) selectdevice**

où *nom*, égal par exemple à **laserjet**, est un code d'imprimante figurant dans la liste du fichier **DEVS.mak**. Bien sûr, GS est limité par la résolution de la machine. A résolution égale, les **dessins** produits sous GS par une HPLaserjet sont identiques à ceux produits par une imprimante PS. Par contre, les **caractères** fournis avec GS n'atteignent pas toujours la qualité de ceux d'Adobe **(2)**. Cette imperfection est peu visible à l'écran (sauf en grande taille), mais elle se manifeste beaucoup plus vite sur le papier.

Quoiqu'il en soit, la firme Aladdin a mis à la libre disposition des informaticiens un outil remarquable qui s'améliore encore de version en version. Il permet à peu de frais de s'initier à PostScript et de mettre au point les dessins et des images les plus complexes beaucoup plus vite qu'avec une imprimante. Rares sont les cas où l'imprimante PostScript reste nécessaire (résolution meilleure, polices plus variées, couleurs plus fidèles dans les images ...).

---

1 - Dans les versions récentes, les fichiers sources (en C) sont distribués à part (fichiers **src.zip**).

2 - La qualité des polices s'améliore à chaque version. Leur liste s'obtient comme il est dit page 68. Il s'agit de celles cataloguées dans le fichier **Fontmap**. Si le fichier police correspondant n'existe pas, GS lui substitue une autre police et affiche un avertissement (à condition que la valeur **false** ait été affectée à la variable **quiet** dans le fichier **GS_INIT.PS**).

GSVIEW

Ghostscript a été conçu plutôt pour représenter des fichiers d'une seule page, tels les **EPSF**. On peut toutefois afficher successivement toutes les pages d'un document en passant à la suivante avec la touche RC *(Enter)*, mais sans pouvoir revenir en arrière.

Ghostview a été écrit par Tim Theisen pour Unix et **Gsview** par Russel Lang pour OS2 et Windows. Ces logiciels, également gratuits, sont distribués largement par Internet en même temps que GhostScript (environ 1 Mo) ; ils affichent à la perfection les documents PS **multipages** pourvu qu'ils soient conformes aux DSC. On peut afficher les pages dans l'ordre croissant (touche +), décroissant (touche −) ou dans un ordre quelconque. On peut les agrandir, les faire pivoter de 90° et enfin imprimer n'importe quelle page − comme avec Ghostscript − sur toute imprimante reconnue par le système d'exploitation, à condition qu'un pilote spécifique ait été écrit pour cette imprimante **(3)**. Tout ceci se commande par *clics* dans une fenêtre *à la Windows*.

Gsview utilise Ghostscript pour interpréter les pages PostScript. Chaque version de Gsview exige une version déterminée de GS. Il existe deux versions "courantes" : l'une est autorisée à tous (distribution **gnu**), l'autre plus récente est restreinte aux utilisations non commerciales (distribution **Aladdin**). L'importation de l'une de ces versions (avec le ou les deux fichiers de police compatibles, cf. p. 72) demande quelque attention. Une page Internet explique comment faire en fonction du système d'exploitation **(4)** : sous Windows, quatre (ou cinq) fichiers comprimés sont nécessaires, trois pour GhostScript, dont un (ou deux) de polices, et un pour Gsview (**gsv***nnnsys*.**zip**).

On décomprime le fichier **gsv***nnnsys*.**zip** : il fournira des fichiers d'explications (**README**, ...) ainsi qu'un exécutable **install.exe** qui va à peu près tout installer. A sa première exécution, Gsview demandera à préciser la configuration (nom du répertoire contenant Ghostscript et celui des polices). Sous Windows, si l'on veut utiliser Ghostscript directement sans passer par Gsview, il faudra initialiser la variable d'environnement **GS_LIB** (fichier **autoexec.bat** sous Dos)

set GS_LIB = *répertoire du fichier* **GS_INI.PS ;** *répertoire des polices* GhostScript (séparer ces noms par un point-virgule).

Avec Gsview (ou Ghostview) on peut afficher à l'écran deux fenêtres réduites, l'une avec un éditeur (ex. Notepad) dans laquelle on écrit ou on modifie le texte du fichier PostScript, l'autre dans laquelle Gsview (après activation et demande de régénération) affiche le dessin programmé. Toutefois, il semble que les diagnostics d'erreur soient plus précis avec Ghostscript qu'avec Gsview **(5)**. C'est pourquoi il paraît utile de pouvoir parfois recourir directement à lui (sous Windows, associer les fichiers PS à Gsview, mais installer une icône pour Ghostscript).

Gsview est capable d'afficher également les fichiers **pdf**. D'autres interpréteurs existent en libre service sur Internet pour représenter des documents PostScript (liste disponible sur Internet) **(6)** ; en général payants, ce sont soit des *partagiciels (freeware)* comme RoPS (30$) ou Printfile ou des produits commerciaux comme SuperPrint. Ils sont beaucoup moins volumineux et moins élaborés que GhostScript (ils utilisent en général les polices de Windows).

Le site *primaire* relatif à Ghostscript est le serveur **www.cs.wisc.edu/~ghost**, mais de nombreux autres serveurs plus proches de nous rediffusent son contenu dans leurs répertoires (sites *miroirs*, comme celui de Paris-Jussieu).

Une abondante "littérature" existe sur Internet au sujet de Gsview, Ghostscript ou PostScript ; elle témoigne de la vitalité du langage et de l'intérêt de ses interpréteurs gratuits. On pourra consulter :

- *Introduction to Ghostscript* (www.cs.wisc.edu/~ghost/intro) ;
- *GSview help* (www.cs.wisc.edu/~ghost/gsview/winhelp) ;
- *GhostScript user manual*, de Thomas Mertz, 24 pages (sites GS) ;
- *Ghostscript, Ghostview and GSview* (www.cs.wisc.edu/~ghost) ;
- *PostScript sins*, par Kevin Thompson (www.byte.com/art/9508/sec13/art3).
- *Gripes*, par Russel Lang (www.cs.wisc.edu/~ghost/gripes) ;
- *What is PostScript* ? (www.gkss.de/W3/PS)
- *Frequently asqued questions* (ou FAQ) (www.lib.ox.ac.uk/internet/news/faq/archive/postscript.faq)

3 - liste sur Internet dans **www.cs.wisc.edu/~ghost/printer.** Il semble cependant que les pilotes **mswinpr2** ou **uniprint** soient particulièrement recommandés.
4 - http:\\www.cs.wisc.edu/~ghost/aladdin/get510 (en 98).

5 - Pour voir les messages d'avertissement et le résultat des ordres **pstack** ou ==, menu *fichier, afficher les messages*.
6 - www.amtec.com/notes/psconv.