Un manuel de l'utilisateur pour METAPOST*

John D. HOBBY

Bell Labs, Lucent Technologies
Murray Hill, NJ 07974

Résumé. Le système METAPOST implémente un langage de construction de figures ressemblant beaucoup au METAFONT de KNUTH sauf qu'il produit des commandes POSTSCRIPT au lieu de *bitmaps* codés. METAPOST est un langage puissant de production de figure pour les documents devant être imprimés sur des imprimantes POSTSCRIPT. Il fournit un accès facile à toutes les fonctionnalités de POSTSCRIPT et rend aisée l'intégration de textes et de graphiques.

Ce manuel d'introduction pour l'utilisateur ne requiert pas de connaissances de METAFONT ni la possession du *METAFONTbook* mais l'un ou l'autre sera bénéfique. Un appendice explique les différences entre METAPOST et METAFONT.

Abstract. *The METAPOST system implements a picture-drawing language very much like Knuth's METAFONT except that it outputs* POSTSCRIPT *commands instead of run-length-encoded bitmaps. METAPOST is a powerfull language for producing figures for documents to be printed on* POSTSCRIPT *printers. It provides easy access to all the features of* POSTSCRIPT *and it facilities for integrating text and graphics.*

This document serves as an introductory user's manual. It does not require knowledge of METAFONT or access to The METAFONTbook, but both are beneficial. An appendix explains the differences between METAPOST and METAFONT.

1. Introduction

METAPOST est un langage de programmation très proche du langage METAFONT de KNUTH [4], la différence essentielle étant que METAPOST produit un fichier POSTSCRIPT alors que METAFONT produit un fichier *bitmap*. Les outils de base pour la création et la manipulation de graphiques ont tous été empruntés à METAFONT. Ils incluent les types numériques, les paires de coordonnées, les splines cubiques, les transformations affines, les chaînes de texte et les quantités booléennes. Quelques propriétés ont été ajoutées facilitant l'intégration de textes et de graphiques et accédant aux

* Ce manuel est paru en anglais sous le titre *A User's manual for METAPOST*. Il est publié ici avec l'aimable autorisation de l'auteur. Traduction française de Pierre FOURNIER (pierre.fournier@unilim.fr) et Jean-Côme CHARPENTIER (jean-come.charpentier@wanadoo.fr)

propriétés spécifiques de POSTSCRIPT comme le détourage, l'estompage et les lignes en pointillés. Une autre caractéristique empruntée à METAFONT est la possibilité de résoudre les équations linéaires données sous forme implicite ; cela permet à de nombreux programmes d'être écrits dans un style largement déclaratif. En construisant des opérations complexes à partir d'opérations simples, METAPOST apporte à la fois puissance et flexibilité.

METAPOST est particulièrement bien adapté à la production de figures et de documents techniques dont certains aspects peuvent être contrôlés par des contraintes géométriques ou mathématiques qui sont mieux exprimées sous forme symbolique. En d'autres termes, METAPOST n'est pas destiné à remplacer un éditeur de dessins ni un éditeur graphique interactif, il est réellement un langage de programmation destiné plus particulièrement à produire des figures devant être incluses dans des documents TEX ou troff. Les dessins obtenus peuvent être inclus dans un document TEX à partir du programme *freeware* `dvips` comme le montre la figure 1.

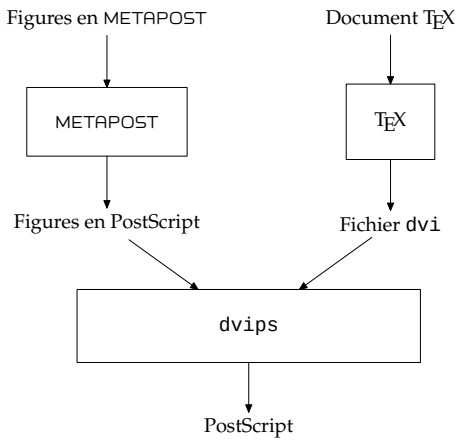


FIG. 1 – Un diagramme du processus pour un document TEX contenant des figures en METAPOST

Pour utiliser METAPOST, on doit préparer un fichier source contenant les commandes METAPOST puis appeler METAPOST, normalement en donnant une commande de la forme :

`mp`<nom-fichier>

Cette syntaxe peut dépendre du système. Théoriquement, les fichiers source pour METAPOST ont une extension « `.mp` » ce qui permet de ne pas préciser l'extension lorsqu'on appelle METAPOST. Ainsi, pour un fichier source s'appelant `foo.mp`, l'appel se fera sous la forme :

```
mp foo
```

Cet appel produira des fichiers de sortie appelés `foo.1` et `foo.2`. Toutes les entrées/sorties du terminal sont enregistrées dans un fichier de transcription appelé `foo.log`. Ce fichier de transcription comporte les messages d'erreur et toutes les commandes METAPOST entrées de manière interactive[1]. Le fichier de transcription débute par une ligne d'en-tête qui indique quelle version de METAPOST est en cours d'utilisation.

Ce manuel est une introduction au langage METAPOST ; il commence avec les caractéristiques les plus simples à utiliser et les plus importantes applications. Les premiers chapitres décrivent le langage tel qu'il apparaît aux utilisateurs novices avec les paramètres clés ainsi que leurs valeurs par défaut. Certaines fonctionnalités décrites dans ces chapitres font partie d'un ensemble de macros appelé Plain. Les chapitres suivants résument le langage complet et réalise la distinction entre les primitives et les macros pré-chargées de l'ensemble Plain. Comme beaucoup d'éléments de ce langage sont communs avec le METAFONT de KNUTH, on trouvera en appendice une comparaison détaillée des deux langages afin que les utilisateurs expérimentés puisse en connaître plus sur METAPOST en lisant le *METAFONTbook* [4].

2. Déclarations élémentaires de dessin

Les déclarations de dessin les plus simples sont celles qui engendrent des lignes droites. Ainsi

```
draw (20,20) -- (0,0);
```

trace une ligne diagonale et

```
draw (20,20) -- (0,0) -- (0,30) -- (30,0) -- (0,0);
```

trace une ligne polygonale comme celle-ci :

Que signifient des coordonnées comme `(30,0)` ? METAPOST utilise, par défaut, le même système de coordonnées que POSTSCRIPT. Cela signifie que `(30,0)` correspond à 30 unités à droite de l'origine, où l'unité est $\frac{1}{72}$ de pouce. On se référera à cette unité de mesure par défaut comme un *point* POSTSCRIPT pour le distinguer du point

1 Un message d'invite « * » est utilisé pour les entrées interactives et le message d'invite « ** » indique qu'un nom de fichier source est attendu. Tout ceci peut être évité en appelant METAPOST sur un fichier se terminant par la commande `end`.

d'impression courant qui vaut $\frac{1}{72{,}27}$ de pouce.

METAPOST utilise les mêmes noms d'unités de mesure que TEX et METAFONT . Ainsi `bp` *(big point)* correspond au point POSTSCRIPT et `pt` au point d'impression. D'autres unités de mesure incluent `in` pour pouce, `cm` pour centimètre et `mm` pour les millimètres. Par exemple,

```
draw (2cm,2cm) -- (0,0) -- (0,3cm) -- (0,0);
```

crée une version plus grande de la figure précédente. Écrire `0` équivaut à `0cm` parce que `cm` est en fait un simple facteur de conversion et `0cm` multiplie le facteur de conversion par zéro. METAPOST comprend donc les constructions telles que `2cm` comme un raccourci de `2*cm`.

Il est souvent commode d'introduire son propre facteur d'échelle, soit u. On peut alors définir des coordonnées en fonction de u et décider plus tard si l'on veut commencer avec `u=1cm` ou `u=0.5cm`. Cela donne à l'utilisateur le contrôle de ce qui doit être mis à l'échelle et de ce qui ne doit pas l'être de telle sorte que le changement de u n'affectera pas les éléments tels que l'épaisseur des lignes.

Il y a plusieurs façons de modifier l'apparence d'une ligne en plus du changement de son épaisseur. Les mécanismes du contrôle d'épaisseur permettent beaucoup de généralités dont nous n'avons pas besoin maintenant. Cela conduit à des déclarations telles que

```
pickup pencircle scaled 4pt
```

pour établir à 4 points l'épaisseur de la ligne de la déclaration `draw` suivante. (Cela correspond environ à huit fois l'épaisseur par défaut d'une ligne).

Avec une telle épaisseur de ligne, même une ligne de longueur zéro apparaît comme un gros point gras. On peut utiliser cette déclaration pour faire une grille de points gras en ayant une déclaration `draw` pour chaque point de la grille. Une telle séquence répétitive de déclaration `draw` est plus judicieusement écrite sous la forme de deux boucles imbriquées :

```
for i=0 upto 2 :
   for j=0 upto 2 : draw (i*u,j*u); endfor
endfor
```

La boucle externe fonctionne pour $i = 0, 1, 2$ et la boucle intérieure pour $j = 0, 1, 2$. Le résultat est une grille carrée de 9 points gras comme le montre la figure 2. La figure inclut aussi une version plus grande de la ligne polygonale vue précédemment.

```
beginfig(2);
u=1cm;
draw (2u,2u) -- (0,0) --
(0,3u) -- (3u,0) -- (0,0);
pickup pencircle scaled 4pt;
for i=0 upto 2 :
for j=0 upto 2 : draw
(i*u,j*u); endfor
endfor
endfig;
```

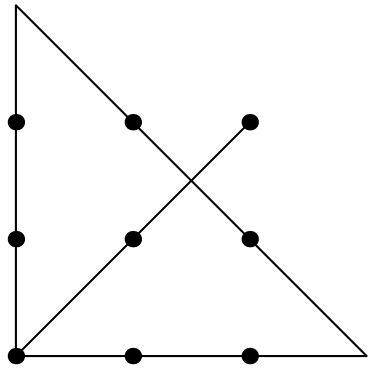


FIG. 2 – Commandes METAPOST et la sortie résultante

Le programme de la figure 2 commence avec `beginfig(2)` et finit avec `endfig`. Ce sont des macros qui réalisent différentes fonctions « administratives » et assurent que les résultats de toutes les déclarations `draw` soient groupés et traduits en POSTSCRIPT. Un fichier source METAPOST contient normalement une succession de paires `beginfig`, `endfig` avec une déclaration `end` après la dernière. Si le fichier est nommé `fig.mp`, le résultat à partir des déclarations `draw` contenues entre `beginfig(1)` et le `endfig` suivant est écrit dans le fichier `fig.1`. En d'autres termes, l'argument numérique de la macro `beginfig` détermine le nom du fichier de sortie correspondant.

Que faire de tous les fichiers POSTSCRIPT ? Ils peuvent être inclus comme figures dans un document TEX ou troff si l'on possède un pilote de sortie qui gère des figures EPS. Si le répertoire standard de macros TEX contient un fichier `epsf.tex`, on peut vraisemblablement inclure la `fig.1` dans un document TEX de la façon suivante :

```
\input epsf
          ⋮
$$\epsfbox{fig.1}$$
```

La macro `epsfbox` calcule l'espace à libérer pour la figure et utilise la commande `\special` de TEX pour insérer la figure `fig.1` souhaitée.

Il est aussi possible d'inclure une production de METAPOST dans un document troff. Le package `-mfpictures` définit une commande `.BP` qui inclut un fichier POSTSCRIPT encapsulé. Par exemple, la commande troff

```
.BP fig.1 3c 3c
```

inclut la figure `fig.1` et spécifie que sa hauteur et sa largeur sont toutes deux en centimètres.

3. Courbes

METAPOST est tout aussi capable de tracer des courbes que des lignes droites. Une déclaration `draw` avec des points séparés par `..` trace une courbe passant par les points. Considérons, par exemple,

```
draw z0..z1..z2..z3..z4
```

après avoir défini les cinq points comme suit :

```
z0 = (0,0); z1 = (60,40); z2 = (40,90);
      z3 = (10,70); z4 = (30,50);
```

La figure 3 montre le résultat : une courbe passant par les points `z0` à `z4`.

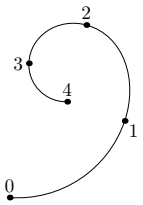


FIG. 3 – Le résultat de `draw z0..z1..z2..z3..z4`

Il existe beaucoup d'autres manières pour tracer un chemin courbe passant par les cinq mêmes points. Pour obtenir un chemin fermé, il faut connecter `z4` au point `z0` en ajoutant `..cycle` à la déclaration `draw` (cf. figure 4). Il est possible de mélanger dans une même déclaration `draw` des lignes courbes et des lignes droites. Il suffit d'utiliser `--` pour obtenir des lignes droites et `..` pour des lignes courbes.

Ainsi,

```
draw z0..z1..z2..z3 -- z4 -- cycle
```

produit une ligne courbe passant par les points 0, 1, 2, 3 et des segments reliant les points 3, 4 et 0. Ce résultat aurait pu être obtenu par les deux déclarations suivantes :

```
draw z0..z1..z2..z3
```

et

```
draw z3 -- z4 -- z0
```

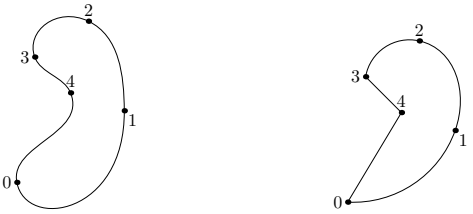


FIG. 4 – (a) : le résultat de `draw z0..z1..z2..z3..z4..cycle;` (b) : le résultat de `draw z0..z1..z2..z3 -- z4 -- cycle`

3.1. Courbes de BÉZIER cubiques

Lorsqu'on demande à METAPOST de tracer une courbe passant par une série de points, METAPOST construit une courbe cubique par morceaux *(splines)* à pente continue et à courbure sensiblement constante (sans variation brusque de courbure). Cela signifie qu'une déclaration de chemin telle que

```
z0..z1..z2..z3..z4
```

conduit à une courbe qui peut être définie paramétriquement par $(X(t),\ Y(t))$ pour $0 \leq t \leq 5$, où $X(t)$ et $Y(t)$ sont des fonctions cubiques par morceaux. Si `z0`=$(x_0,\ y_0)$, `z1`=$(x_1,\ y_1)$, `z2`=$(x_2,\ y_2)$,... METAPOST sélectionne les points de contrôle $(x_0^+,\ y_0^+)$, $(x_1^-,\ y_1^-)$, $(x_1^+,\ y_1^+)$,... pour lesquels

$$\begin{aligned} X(t+i) &= (1-t)^3 x_i + 3t(1-t)^2 x_i^+ + 3t^2(1-t)x_{i+1}^- + t^3 x_{i+1}, \\ Y(t+i) &= (1-t)^3 y_i + 3t(1-t)^2 y_i^+ + 3t^2(1-t)y_{i+1}^- + t^3 y_{i+1} \end{aligned}$$

pour $0 \leq t \leq 1$

Les règles précises pour choisir les points de contrôle sont décrites dans [2] et dans le *METAFONTbook* [4].

Pour que le chemin ait des pentes continues au point $(x_i,\ y_i)$, les tangentes à gauche et à droite en $(X(i),\ Y(i))$ doivent coïncider. Ainsi les vecteurs

$$(x_i - x_i^-,\ y_i - y_i^-) \quad \text{et} \quad (x_i^+ - x_i,\ y_i^+ - y_i)$$

doivent avoir la même direction ; c'est-à-dire qu'ils doivent être sur le segment entre $(x_1^-,\ y_1^-)$ et $(x_1^+,\ y_1^+)$.

Cette situation est illustrée figure 5. Dans cette figure, les points de contrôle sélectionnés par METAPOST sont reliés par une ligne pointillée. Pour ceux qui sont familiarisés avec les propriétés intéressantes de cette construction, METAPOST permet de spécifier les points de contrôle directement avec le format suivant.

```
draw (0,0)..controls(26.8,-1.8) and (51.4,14.6)
..(60,40)..controls(67.1,61.0) and (59.8,84.6)
..(40,90)..controls(25.4,94.0) and (10.5,84.5)
..(10,70)..controls( 9.6,58.8) and (18.8,49.6)
..(30,50);
```

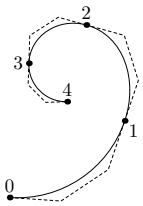


FIG. 5 – Le résultat de `draw z0..z1..z2..z3..z4` avec le polygone (en pointillé) des points de contrôle

3.2. Spécification de direction, tension et courbure

METAPOST permet de contrôler, de plusieurs façons, le comportement d'un chemin courbe sans spécifier réellement les points de contrôle. Par exemple, des points du chemin peuvent être sélectionnés comme des extremums verticaux ou horizontaux. Si `z1` doit être un point à tangente verticale et `z2` un point à tangente horizontale, on peut spécifier que cette tangente en $(X(t),\ Y(t))$ doit être ascendante en `z1` et vers la gauche en `z2` par

```
draw z0..z1{up}..z2{left}..z3..z4;
```

La courbe (figure 6) possède bien les directions verticale et horizontale souhaitées aux points `z1` et `z2`, mais elle n'apparaît pas aussi « douce » que le courbe de la figure 3. La raison est la grande discontinuité de courbure en `z1`. Si la direction en `z1` n'avait pas été spécifiée, METAPOST aurait choisi une direction permettant une courbure sensiblement identique de part et d'autre de `z1`.

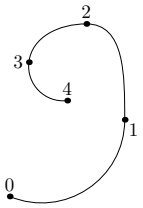
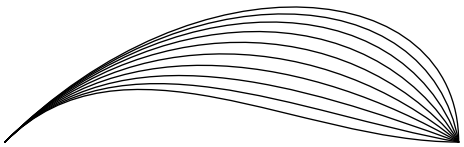


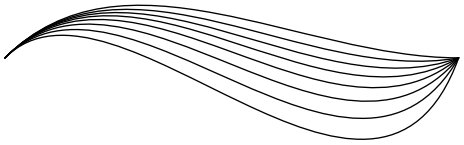
FIG. 6 – Le résultat de `draw z0..z1{up}..z2{left}..z3..z4`

Comment le choix des directions en des points donnés de la courbe détermine-t-il si la courbure sera sans variations brusques ou pas ? La raison est que les courbes utilisées par METAPOST proviennent d'une famille dans laquelle un chemin est déterminé par ses points de début et de fin ainsi que les directions de ces points. Les figures 7 et 8 donnent une bonne idée de ce que sont ces familles de courbes.



```
beginfig(7)
for a=0 upto 9 :
  draw (0,0){dir 45} ..
{dir -10a}(6cm,0);
endfor
endfig;
```

FIG. 7 – Une famille de courbes et les instructions METAPOST associées



```
beginfig(8)
for a=0 upto 7 :
  draw (0,0){dir 45} ..
{dir 10a}(6cm,0);
endfor
endfig;
```

FIG. 8 – Une autre famille de courbes et les instructions METAPOST associées

Les figures 7 et 8 illustrent quelques nouvelles caractéristiques de METAPOST. La première est l'opérateur `dir` qui requiert un angle en degrés et qui crée un vecteur unitaire dans cette direction. Ainsi, `dir 0` est équivalent à `right` et `dir 90` à `up`. Il existe également les vecteurs directions prédéfinis `left` et `down` pour `dir 180` et `dir 270`.

Les vecteurs directions donnés entre `{ }` peuvent être de n'importe quelle longueur. Ils peuvent être placés aussi bien avant un point qu'après. Il est même possible de spécifier une direction avant *et* après un point. Par exemple une indication de la forme

```
..{dir 60}(10,0){up}..
```

produira une courbe possédant un coin en (10, 0).

Il faut noter que certaines courbes de la figure 7 ont des points d'inflexion. Ceci est nécessaire de façon à produire des courbes lisses dans des situations telles que dans la figure 4a, mais ce n'est probablement pas souhaitable quand les points extrêmes ont des tangentes horizontales ou verticales comme dans la figure 9a. Pour que `z1` soit le point le plus haut sur la courbe (cf. figure 9b), il suffit d'utiliser `...` à la place de `..` (deux points). La commande `...` (trois points) permet d'obtenir une courbe sans

point d'inflexion sauf si certaines contraintes l'interdisent. (Il serait possible d'éviter les inflexions dans la figure 7, mais pas dans la figure 8)

(a) `draw z0{up} .. z1{right} .. z2{down}`

(b) `draw z0{up} ... z1{right} ... z2{down}`

FIG. 9 – Deux déclarations `draw` et les courbes résultantes

Une autre manière de contrôler le « mauvais » comportement d'une courbe est d'augmenter le paramètre `tension` . La commande `..` met le paramètre `tension` à sa valeur par défaut égale à 1. Si cette valeur fait apparaître une courbe un peu trop brusque, il est possible d'augmenter sélectivement la tension. Si la courbe de la figure 10a est considérée « trop brusque », une déclaration `draw` sous la forme suivante augmente la tension entre les points `z1` et `z2` :

```
draw z0..z1..tension 1.3..z2..z3
```

La figure 10b montre le résultat. Pour un effet asymétrique, comme sur la figure 10c, la déclaration `draw` devient :

```
draw z0..z1..tension 1.3 and 1..z2..z3
```

Le paramètre tension peut être inférieur à 1 mais il doit être au moins égal à $\frac{3}{4}$.

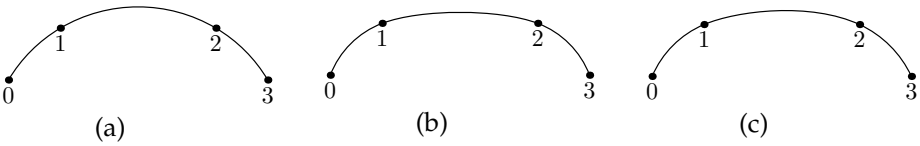


FIG. 10 – Résultats de `draw z0..z1..tension` α `and` β `..z2..z3` pour différentes valeurs de α et β : (a) $\alpha = \beta = 1$; (b) $\alpha = \beta = 1.3$; (c) $\alpha = 1.5$, $\beta = 1$

Les chemins METAPOST possèdent aussi un paramètre appelé `curl` (ou courbure) qui affecte les extrémités d'un chemin. En l'absence de toute spécification de direction, le premier et le dernier segment d'un chemin non fermé sont approximativement des arcs circulaires comme dans le cas $c = 1$ de la figure 11. Pour utiliser des valeurs différentes pour le paramètre `curl`, on spécifie cette valeur par la commande `{curl c}`. Ainsi

```
draw z0{curl c}..z1..{curl c}z2;
```

fixe le paramètre curl pour `z0` et `z2`. De faibles valeurs du paramètre `curl` réduisent la courbure au point extrême du chemin, alors que des valeurs importantes augmentent la courbure (cf. figure 11). En particulier, une valeur nulle engendre une courbure proche de zéro.

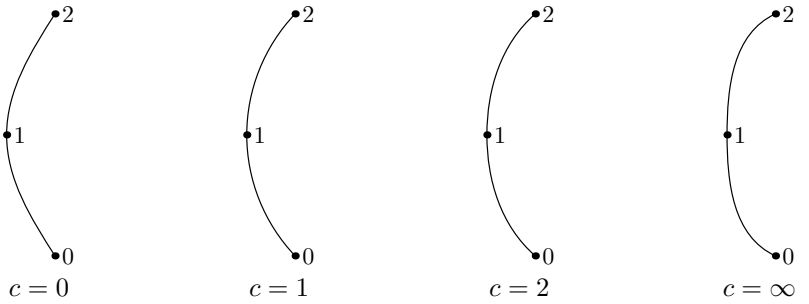


FIG. 11 – Résultats de `draw z0{curl c} .. z1 .. {curl c}z2` pour différentes valeurs du paramètre `curl` c

3.3. Résumé de la syntaxe des chemins

Il existe d'autres caractéristiques de la syntaxe des chemins METAPOST mais elles sont relativement peu importantes. Puisque METAFONT utilise la même syntaxe des chemins, on pourra se référer au *METAFONTbook* [4]. Le résumé de la syntaxe, figure 12, inclut tout ce qui a été discuté jusqu'ici, y compris les constructions `--` et `...` que le *METAFONTbook* considère plus comme des macros que comme des primitives [4].

Quelques commentaires d'ordre sémantique sont utiles ici : s'il existe un <spécificateur de direction> non vide avant un <nœud de chemin> mais pas après celui-ci, ou vice versa, la direction spécifiée (ou la quantité de `curl`) s'applique aux deux segments d'arrivée et de départ du chemin. Un arrangement semblable s'applique quand une spécification de <contrôle> ne donne qu'une <paire primaire>. Ainsi

```
..controls (30,20)..
```

est équivalent à

```
..controls (30,20) and (30,20)..
```

Un couple de coordonnées comme `(30,20)` ou une variable `z` est ce que la figure 12 nomme une <paire primaire> (<pair primary>).

Un <nœud de chemin> (<path knot>) est semblable sauf qu'il peut prendre une autre forme telle qu'une expression de chemin entre parenthèses. Primaires et expressions de types variés seront étudiés en toute généralité dans la section 5.

<expression chemin>→ <sous expression chemin>
  | <sous expression chemin><spécificateur de direction>
  | <sous expression chemin><jointure de chemin>`cycle`
<sous expression chemin>→ <nœud de chemin>
  | <sous expression chemin><jointure de chemin><nœud de chemin>
<jointure de chemin>→ `--`
  | <spécificateur de direction><jointure de chemin élémentaire><spécificateur de direction>
<spécificateur de direction>→ <vide>
  | <`{curl`<expression numérique>`}`
  | <`{`<expression paire>`}`
  | <`{`<expression numérique>`,`<expression numérique>`}`
<jointure de chemin élémentaire>→ <`..`|`...`|`..`<tension>`..`|`..` <contrôles>`..`
<tension>→`tension`<primaire paire>
  |`tension`<numérique primaire>`and`<numérique primaire>
<contrôles>→`controls`<primaire paire>
  |`controls`<paire primaire>`and`<primaire paire>

FIG. 12 – Syntaxe de la construction des chemins

4. Équations linéaires

Une caractéristique importante, empruntée à METAFONT, est la possibilité de résoudre des équations linéaires de telle sorte que des programmes peuvent être écrits en partie de façon déclarative.

Par exemple, l'interpréteur METAPOST peut lire

```
a+b = 3; 2*a = b+3;
```

et en déduire que $a = 2$ et que $b = 1$. La même équation peut être écrite de façon nettement plus compacte sous la forme :

```
a+b = 2*a-b = 3;
```

Quelle que soit la forme choisie, la commande `show a, b;` permet de visualiser les valeurs de `a` et de `b`. METAPOST répond en affichant :

```
» 2
» 1
```

Il faut noter que = n'est pas un opérateur d'affectation. Il déclare simplement que le membre gauche égale le membre droit. Ainsi, `a=a+1` produit un message d'erreur se plaignant d'une « équation inconsistante » [2] *(Inconsistent equation)*. La manière d'incrémenter la valeur de `a` est d'utiliser un opérateur d'affectation `:=` :

```
a :=a+1;
```

En d'autres termes, `:=` permet de changer une valeur existante alors que = est destiné à la résolution d'équations linéaires.

Il n'y a aucune restriction en ce qui concerne les mélanges d'affectations et d'équations, comme le montre l'exemple suivant :

```
a = 2; b = a; a := 3; c = a;
```

Les deux premières équations posent `a` et `b` égales à 2, l'opérateur d'affectation change la valeur de `a` à 3 sans modifier `b`. Le résultat final de `c` est 3 puisque `c` est mis en équation comme étant égal à la nouvelle valeur de `a`. En général, une opération d'affectation est interprétée en calculant d'abord la nouvelle valeur, puis en éliminant l'ancienne valeur à partir de toutes les équations existantes avant d'affecter réellement la nouvelle valeur.

4.1. Équations et couples de coordonnées

METAPOST peut aussi résoudre des équations linéaires mettant en jeu des couples de coordonnées. Des exemples simples ont déjà été vus sous la forme d'équations du type

```
z1=(0,.2in);
```

Chaque membre de l'équation doit être constitué d'addition ou de soustraction de couples de coordonnées et de leur multiplication ou division par une quantité numérique connue. D'autres façons de nommer des variables de type paire seront étudiées plus loin, mais la forme `z`<nombre> est pratique car c'est une abréviation pour

(`x`<nombre>, `y`<nombre>)

Cela permet d'affecter des valeurs à des variables `z` par l'intermédiaire d'équations mettant en jeu leurs coordonnées. Par exemple, les points `z1`, `z2`, `z3` et `z6` de la figure 13 ont été initialisés par les équations suivantes :

[2] [Ndt] : Une équation inconsistante est une équation qui n'a pas de solution (cf. *METAFONTBook* [4, page 82]).

```
z1=-z2=(.2in,0);
x3=-x6=.3in;
x3+y3=x6+y6=1.1in;
```

Les mêmes points peuvent être obtenus en indiquant directement leurs valeurs :

```
z1=(.2in,0);     z2=(-.2in,0);
z3=(.3in,.6in); z6=(-.3in,1.2in);
```

Après avoir lu les équations, l'interpréteur de METAPOST connaît les valeurs de `z1`, `z2`, `z3` et `z6`. L'étape suivante dans la construction de la figure 13 est de définir les points `z4` et `z5` équi-répartis le long du segment `z3z6`.

Puisque cette opération intervient souvent, METAPOST offre une syntaxe particulière pour cela.

Cette construction d'un point intermédiaire *(mediation construction)*

```
z4=1/3[z3,z6];
```

signifie que `z4` est situé au $\frac{1}{3}$ du segment `z3z6` ; *i.e.*

$$\texttt{z4}=\texttt{z3}+\tfrac{1}{3}(\texttt{z6-z3})$$

De façon similaire

```
z5=2/3[z3,z6];
```

signifie que `z5` est situé au $\frac{2}{3}$ du segment `z3z6`.

Cet opérateur peut aussi être utilisé pour dire qu'un point se situe quelque part entre deux points connus. Pour l'instant, on introduit une nouvelle variable `aa` et on écrit quelque chose comme :

```
z20=aa[z1,z3];
```

Cela signifie que `z20` est une certaine fraction `aa` inconnue sur le segment entre `z1` et `z3`. Une autre équation de ce type faisant intervenir un autre segment est suffisante pour fixer la valeur de `z20`. Pour dire que `z20` est l'intersection du segment `z1-z3` et du segment `z2-z4`, il faut introduire une autre variable `ab` et poser

```
z20=ab[z2,z4];
```

Ceci permet à METAPOST de résoudre `x20`, `y20`, `aa` et `ab`.

Il peut être un peu pénible de devoir se souvenir de nouveaux noms tels que `aa` et `ab`. On peut éviter cette peine en utilisant une variable spéciale appelée `whatever`. Une nouvelle variable anonyme est créée chaque fois que cette macro est employée.

Ainsi la déclaration

```
beginfig(13);
z1=-z2=(.2in,0);
x3=-x6=.3in;
x3+y3=x6+y6=1.1in;
z4=1/3[z3,z6];
z5=2/3[z3,z6];
z20=whatever[z1,z3]=whatever[z2,z4];
z30=whatever[z1,z4]=whatever[z2,z5];
z40=whatever[z1,z5]=whatever[z2,z6];
draw z1 -- z20 -- z2 -- z30
-- z1 -- z40 -- z2;
pickup pencircle scaled 1pt;
draw z1 -- z2; draw z3 --
z6;
endfig;
```

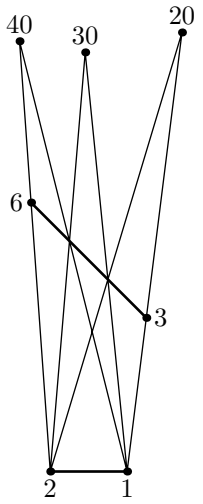


FIG. 13 – Commandes METAPOST et la figure résultante. Le numéro des points a été ajouté pour plus de clarté

```
z20=whatever[z1,z3]=whatever[z2,z4];
```

détermine `z20` comme précédemment, mis à part le fait qu'elle utilise `whatever` pour créer deux variables anonymes *différentes* plutôt que `aa` et `ab`. C'est ainsi que la figure 13 détermine `z20`, `z30` et `z40`.

4.2. Gérer les inconnues

Un système d'équations telles que celles utilisées dans la figure 13 peut être donné dans n'importe quel ordre tant que les équations sont linéaires et que toutes les variables sont déterminées avant d'être utilisées. Cela signifie que les équations

```
z1=-z2=(.2in,0);
x3=-x6=.3in;
x3+y3=x6+y6=1.1in;
z4=1/3[z3,z6];
z5=2/3[z3,z6];
```

suffisent à déterminer les variables `z1` jusqu'à `z6`, quel que soit l'ordre des équations.

D'autre part, l'expression

```
z20=whatever[z1,z3];
```

n'est légale que si la différence `z3-z1` est connue, car l'exigence de la linéarité ne permet pas de multiplier une composante inconnue de `z3-z1` par la valeur anonyme et inconnue de `whatever`. La règle générale est qu'on ne peut pas multiplier deux quantités inconnues ou diviser par une quantité inconnue. De même, une quantité inconnue ne peut pas être employée dans une déclaration `draw`. Puisque les équations linéaires sont autorisées, l'interpréteur METAPOST peut résoudre facilement les équations et reconnaître les valeurs connues.

Le moyen le plus naturel pour s'assurer que METAPOST peut traiter une expression telle que

```
whatever[z1,z3]
```

est de s'assurer que `z1` et `z3` sont connues toutes les deux. Quoiqu'il en soit, cela n'est pas réellement nécessaire puisque METAPOST peut être capable de déduire une valeur pour `z3-z1` avant que `z3` et `z1` soit connues. Par exemple, METAPOST acceptera les équations

```
z3=z1+(.1in,.6in); z20=whatever[z1,z3];
```

mais il sera incapable de déterminer l'une des composantes de `z1`, `z3` ou `z20`.

Ces équations ne donnent que des informations partielles sur `z1`, `z3` et `z20`. Une bonne manière de s'en rendre compte est de poser une autre équation du genre

```
x20-x1=(y20-y1)/6;
```

cela produira le message d'erreur « `! Redundant equation` » [3]. METAPOST suppose que l'utilisateur essaye d'écrire quelque chose de nouveau, il est donc normal d'obtenir un message si l'utilisateur pose une équation redondante. Si la nouvelle équation avait été

```
(x20-x1)=(y20-y1)/6=1in;
```

le message aurait été « `!Inconsistent equation (off by 71.99979)` » (équation inconsistante par erreur d'arrondi). Ce message d'erreur illustre l'erreur d'arrondi dans le mécanisme de résolution des équations linéaires de METAPOST. Les erreurs d'arrondi ne sont normalement pas des problèmes sérieux. Mais c'est généralement la cause de problèmes si l'on essaye quelque chose comme trouver l'intersection de deux lignes qui sont presque parallèles.

---

3 [Ndt] :Équation redondante, une équation redondante est une équation qui n'apporte rien de nouveau et qui est inutile dans le système à résoudre.

5. Expressions

Il est maintenant temps d'avoir une vue plus systématique du langage METAPOST. On a vu qu'il existe des quantités numériques et des couples de coordonnées et que ces éléments peuvent être combinés pour spécifier des chemins dans les déclarations `draw`. On a également vu comment des variables peuvent être utilisées dans des équations linéaires, mais on n'a pas étudié tous les opérateurs et tous les types de variable qui peuvent être employés dans les équations.

Il est possible de faire des essais avec des expressions faisant intervenir n'importe quel type de variable mentionné plus loin en utilisant la déclaration

`show`<expression>

pour demander à METAPOST d'afficher une représentation symbolique de la valeur de chaque expression. Pour des valeurs numériques connues, chacune d'entre elles sera affichée sur une nouvelle ligne précédée par « >> ». D'autres types de résultats sont affichés de façon semblable, exceptées les valeurs complexes qui, parfois, ne sont pas affichées sur des périphériques standards de sortie. Dans ce cas, une référence au fichier de transcription est affichée et ressemble à :

```
>> (picture see the transcript file)
```

Si l'on veut que la totalité des résultats de la commande `show` soit affichée sur l'écran, il faut assigner une valeur positive à la variable interne `tracingonline`.

5.1. Types de données

METAPOST possède en réalité neuf types élémentaires de données : numérique *(numeric)*, couple ou paire *(pair)*, chemin *(path)*, transformation *(transform)*, couleur *(color)*, chaîne de caractères *(string)*, booléen *(boolean)*, figure *(picture)* et stylo ou plume *(pen)*.

5.1.1. Type numérique (numeric)

Les quantités numériques dans METAPOST sont représentées en arithmétique à virgule fixe comme des entiers multiples de $\frac{1}{65536}$. Ces entiers doivent normalement avoir des valeurs absolues inférieures à 4096 mais les résultats intermédiaires peuvent être huit fois plus grands. Cela ne devrait pas poser de problèmes pour les valeurs de distances ou de coordonnées puisque 4096 points POSTSCRIPT représentent plus de 1,4 m. S'il est nécessaire de travailler avec des nombres de valeur absolue supérieure ou égale à 4096, il faut mettre la variable interne `warningcheck` à zéro pour supprimer les messages d'erreurs concernant les quantités numériques importantes.

5.1.2. *Type couple ou paire* (pair)

Le type couple est représenté comme un couple de quantités numériques. On a vu que les couples sont utilisés pour indiquer des coordonnées dans les déclarations `draw`. Les couples peuvent être additionnés, soustraits, utilisés pour déterminer des points intermédiaires *(médiation)* ou encore multipliés ou divisés par des nombres.

5.1.3. *Type chemin* (path)

Les chemins ont déjà été vus précédemment dans le cadre des déclarations `draw`. Cependant il faut noter que les chemins sont des objets de première classe (ou primaires) qui peuvent être stockés et manipulés. Un chemin représente une ligne, droite ou courbe, définie paramétriquement.

5.1.4. *Type transformation* (transform)

Ce type correspond une transformation affine arbitraire. Une transformation peut être une combinaison quelconque de rotation, d'homothétie, de déformation (inclinaison[4]) et de décalage ou de translation. Si `p`=$(p_x,\ p_y)$ est un couple et `T` une transformation,

```
p transformed T;
```

est un couple de la forme

$$(t_x + t_{xx}p_x + t_{xy}p_y,\ t_y + t_{yx}p_x + t_{yy}p_y)$$

où les six quantités $(t_x,\ t_y,\ t_{xx},\ t_{xy},\ t_{yx},\ t_{yy})$ déterminent la transformation `T`. Les transformations peuvent aussi s'appliquer aux chemins *(paths)*, figures *(pictures)*, stylos *(pens)* et aux transformations *(transforms)*.

5.1.5. *Type couleur* (color)

Le type couleur ressemble beaucoup au type couple mais il possède trois composantes au lieu de deux. Comme les couples, les couleurs peuvent être ajoutées, soustraites, utilisées dans des expressions de *médiation*, ou bien être multipliées ou divisées par un nombre. Les couleurs peuvent être spécifiées en termes de constantes prédéfinies `black` (noir), `white` (blanc), `red` (rouge), `green` (vert), `blue` (bleu), ou alors les composantes rouge, verte et bleue peuvent être données explicitement. `black` correspond à `(0,0,0)` et `white` à `(1,1,1)`. Un niveau de gris *(gray)* tel que `(.4,.4,.4)` peut s'écrire `0.4white`. Il n'y a pas de restriction pour définir des couleurs « plus noires que noir » ou « plus blanches que blanc » sauf que toutes les composantes sont ramenées dans l'intervalle [0,1] lorsque la couleur est transmise dans le fichier POSTSCRIPT de sortie. METAPOST résout des équations linéaires mettant en jeu des couleurs de la même façon qu'il le fait avec les couples.

4 `slanting` dans la version originale.

5.1.6. *Type chaîne de caractère* (string)

Les constantes chaînes de caractères sont données entre guillemets simples `"comme ceci"` (ou `"like this"`). Elles ne peuvent pas contenir de guillemets ni de retours à la ligne. Cependant il est possible de construire des chaînes contenant n'importe quelle séquence de caractères codés sur huit bits.

5.1.7. *Type booléen* (boolean)

Le type booléen a deux valeurs `true` (`vrai`) et `false` (`faux`) et les opérateurs `and` (`et`), `or` (`ou`) et `not` (`non`). Les relations `=` et `<>` testent des objets de tout type pour l'égalité et l'inégalité. Les relations de comparaison `<`, `<=`, `>` et `>=` sont définis de façon lexicographique pour les chaînes de caractères et de façon habituelle pour les nombres. Les relations d'ordre sont également définies pour les booléens, les couples, les couleurs et les transformations. Les règles de comparaison ne sont pas discutées ici.

5.1.8. *Type figure* (picture)

Le type figure est simplement ce que son nom indique. Tout ce que METAPOST sait dessiner peut être stocké dans une variable figure *(picture)*. En fait, la déclaration `draw` stocke ses résultats dans une figure spéciale appelée `currentpicture`. Les figures peuvent être ajoutées à d'autres et les transformations peuvent opérer sur elles.

5.1.9. *Type stylo ou plume* (pen)

Enfin, il existe le type stylo ou plume. La principale fonction des plumes (ou stylos), dans METAPOST, est de déterminer l'épaisseur des lignes mais les plumes peuvent être utilisées pour obtenir des effets calligraphiques.

La déclaration

`pickup` <expression stylo>

permet de sélectionner un stylo dans les déclarations `draw` suivantes. Normalement, l'<expression stylo> est de la forme

`pencircle scaled` <primaire numérique>

Ceci définit un stylo circulaire produisant une ligne d'épaisseur constante. Si l'on désire des effets calligraphiques, l'<expression stylo> peut être ajustée pour obtenir une plume elliptique ou polygonale.

5.2. Opérateurs

Il existe beaucoup de manières différentes pour bâtir des expressions à l'aide des neuf types élémentaires, mais la plupart des opérations utilisent une syntaxe relativement simple avec quatre niveaux de priorité comme l'indique la figure 14. Il en résulte quatre types de formules : les formules primaires, secondaires, tertiaires et les expressions de chacun des neuf types élémentaires. Ainsi les règles de syntaxe peuvent être spécialisées pour traiter des éléments tels que <numérique primaire>, <booléen tertiaire>, etc. Ceci permet au type du résultat de l'opération de dépendre du choix de l'opérateur et du type des opérandes. Par exemple, la relation < est une formule <binaire tertiaire> qui peut être appliquée à une <expression numérique> et un <numérique tertiaire> pour donner une <expression booléenne>. Le même opérateur peut accepter d'autres types d'opérandes telles que des <expression chaîne de caractères> et <chaîne de caractères tertiaire>, mais il en résultera un message d'erreur si le type des opérandes ne convient pas.

La multiplication et la division `*` et `/` sont des exemples de ce que la figure 14 appelle un <opérateur binaire primaire>. Chacun accepte deux opérandes numériques ou un opérande de type numérique et un opérande de type paire ou couleur. L'opérateur d'exponentiation `**` est un <opérateur binaire primaire> qui requiert deux opérandes numériques. L'opérateur d'exponentiation est placé au même niveau de priorité que la multiplication ou la division. La conséquence malheureuse est que `3*a**2` signifie $(3a)^2$ et non $3(a)^2$. Puisque l'opérateur unaire de négation s'applique au niveau primaire il s'ensuit que `-a**2` signifie $(-a)^2$.

Heureusement, la soustraction a un niveau de priorité inférieur de telle sorte que `a-b**2` signifie bien $a-(b^2)$ au lieu de $(a-b)^2$.

Un autre <opérateur binaire primaire> (<primary binop>) est l'opérateur `dotprod` qui calcule le produit scalaire de deux couples. Par exemple, `z1 dotprod z2` est équivalent à `x1*x2 + y1*y2`.

Les opérateurs `+` et `-` sont des <opérateurs binaires secondaires> qui opèrent sur des nombres, couples ou couleurs et produisent des résultats de même type. Les autres opérateurs qui entrent dans cette catégorie sont « l'addition pythagoricienne » `++` et la « soustraction pythagoricienne » `+-+` : `a++b` signifie $\sqrt{a^2+b^2}$ et `a+-+b` signifie $\sqrt{a^2-b^2}$. Il existe trop d'opérateurs pour les énumérer tous ici. Signalons quand même deux des plus importants : les opérateurs booléens `and` et `or`. L'opérateur `and` est un <opérateur binaire primaire> et l'opérateur `or` est un <opérateur binaire secondaire>.

Les opérations élémentaires sur les chaînes de caractères *(strings)* sont la concaténation et l'extraction de sous-chaînes *(substring construction)*. L'opérateur `&` <opérateur binaire tertiaire> concatène deux chaînes : `"abc" & "de"` produit la chaîne `"abcde"`.

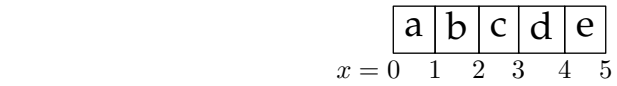
<primaire>→ <variable>
|(<expression>)
| <opérateur nul>
| <opérateur of><expression>`of`<primaire>
| <opérateur unaire><primaire>
<secondaire>→ <primaire>
| <secondaire><opérateur binaire primaire><primaire>
<tertiaire> → <secondaire>
| <tertiaire><opérateur binaire secondaire><secondaire>
<expression>→ <tertiaire>
| <expression><opérateur binaire tertiaire><tertiaire>

FIG. 14 – Syntaxe générale des expressions

L'extraction de sous-chaînes s'obtient par l'opérateur <of> `substring` utilisé comme ceci :

`substring` <expression paire> `of` <chaîne primaire>

L'<expression paire> détermine la partie de la chaîne à sélectionner. À ce propos, la chaîne est indexée de telle façon que les positions entières tombent *entre* les caractères. Le n[e] caractère de la chaîne se trouve entre les « abscisses » $n-1$ et n. Le premier caractère se situe donc entre 0 et 1. Ainsi la chaîne `"abcde"` devra être pensée comme ceci :



et `substring (2,4) of "abcde"` renverra `"cd"`. Il faut un peu de temps pour s'habituer à cette méthode mais cela évite les erreurs peu explicites du genre « off by one ».

Certains opérateurs ne nécessitent aucun argument. Un exemple de ce que la figure 14 appelle un <opérateur nul> est `nullpicture` qui renvoie une figure complètement blanche.

La syntaxe élémentaire, donnée figure 14, ne couvre que les aspects de la syntaxe des expressions qui sont relativement indépendants des types de variables. Par exemple, la syntaxe compliquée de chemins donnée à la figure 12 donne d'autres règles pour une <expression chemin>. Une règle additionnelle <nœud de chemin> → <paire tertiaire> | <paire tertiaire> explique le sens de <nœud de chemin> dans la figure 12.

Cela signifie que l'expression de chemin

```
z1+(1,1){right}..z2
```

ne nécessite pas de parenthèses autour de `z1+(1,1)`.

5.3. Fraction, médiation et opérateurs unaires

Les expressions de médiation n'apparaissent pas dans la figure 14. Les expressions de médiation, du point de vue grammatical, appartiennent au niveau <primaire>. La règle générale de construction est donc :

<primaire>→ <atome numérique> `[`<expression>`,`<expression>`]`

où chaque <expression> peut être de type numérique, couple ou couleur. L'<atome numérique> dans une expression de médiation est le type numérique le plus simple de <primaire numérique> comme le montre la figure 15. Une conséquence de tout cela est que le terme initial, dans une expression de médiation, doit être mis entre parenthèses si ce n'est pas juste une variable, un nombre positif ou une fraction positive. Par exemple,

`-1[a,b]` et `(-1)[a,b]`

sont très différents. La première forme correspond à $-b$ puisque cette forme est équivalente à `-(1[a,b])` ; la seconde correspond à $a-(b-a)$ ou encore $2a-b$.

<numérique primaire>→ <atome numérique>
| <atome numérique>`[`<expression numérique>, <expression numérique>`]`
| <opérateur of>`<expression>of`<primaire>
| <opérateur unaire><primaire>
<atome numérique>→ <variable numérique>
| <nombre ou fraction>
| <`(`<expression numérique>`)`
| <opérateur numérique nul>
<nombre ou fraction>→ <nombre>/<nombre>
| <nombre non suivi de '/<nombre>'>

FIG. 15 – Règles syntaxiques pour les numériques primaires

Une caractéristique remarquable des règles de la figure 15 est que l'opérateur `/` est plus fort lorsque ses opérandes sont des nombres. Ainsi `2/3` est un <atome numérique> alors que `(1+4)/3` n'est seulement qu'un <numérique secondaire>. La différence est plus claire si l'on applique ceci à un <opérateur binaire primaire> tel que `sqrt` :

```
sqrt 2/3
```

signifie $\sqrt{\frac{2}{3}}$ tandis que

```
sqrt (1+1)/3
```

signifie $\sqrt{2}/3$. Des opérateurs tels que `sqrt` peuvent être écrit en notation fonctionnelle standard, mais il est souvent inutile de mettre les arguments entre parenthèses. Cela s'applique à toutes les fonctions à un niveau équivalent à celui d'un <opérateur binaire primaire>. Par exemple, `abs(x)` et `abs x` calculent toutes les deux la valeur absolue de x. Les fonctions `round`, `floor`, `ceiling`, `sind`, `cosd` fonctionnent sur ce même principe. Les deux dernières fonctions calculent le sinus et le cosinus d'un angle exprimé en degrés.

Tous les opérateurs unaires ne nécessitent pas forcément un argument numérique pour renvoyer un résultat numérique. Par exemple, l'opérateur `abs` peut être appliqué à un couple pour calculer la longueur euclidienne d'un vecteur. L'application de `unitvector` à un couple produit ce même couple normé. Sa longueur euclidienne sera ramenée à l'unité. L'opérateur `decimal` transforme un nombre en une chaîne de caractères. L'opérateur `angle` s'applique à un couple (ou vecteur) et calcule l'arctangente ; en d'autres termes, `angle` est l'opérateur inverse de `dir` qui a été vu dans le paragraphe 3.2. Il existe également un opérateur `cycle` qui prend un <chemin primaire> et renvoie un résultat booléen indiquant si le chemin est fermé ou non.

Il existe une classe entière d'autres opérateurs qui prennent des expressions en argument et renvoient des résultats booléens. Un nom de type tel que `pair` peut opérer sur n'importe quel type de <primaire> et renvoie un booléen indiquant si l'argument est de type `pair`. De façon similaire, chacun des types suivant peut être utilisé comme opérateur unaire : `numeric` (numérique), `boolean` (booléen), `color` (couleur), `string` (chaîne de caractères), `transform` (transformation), `path` (chemin), `pen` (stylo) et `picture` (dessin). En plus du simple test de type d'un <primaire>, on peut utiliser les opérateurs `known` et `unknown` pour tester si un <primaire> possède une valeur complètement connue.

Même un nombre peut se comporter comme un opérateur dans certains cas. Ceci fait intervenir une astuce qui permet d'écrire `3x` et `3cm` comme alternative à `3*x` et `3*cm`. La règle est que un <nombre ou fraction> qui n'est pas suivi par `+`, `-` ou un autre <nombre ou fraction> peut servir de <opérateur binaire primaire>. Ainsi `2/3x`

correspond à deux tiers de `x` mais `(2)/3x` vaut $\frac{2}{3x}$ et `3 3` est illégal.

Il existe aussi des opérateurs pour extraire des champs numériques des couples, couleurs et même des transformations. Si `p` est une <paire primaire>, `xpart p` et `ypart p` extraient ses composantes de telle sorte que

```
(xpart p, ypart p)
```

est équivalent à `p` même si `p` est une paire inconnue utilisée dans une équation linéaire. De la même façon, une couleur `c` est équivalente à :

```
(redpart c, greenpart c, bluepart c)
```

L'application de cette commande aux transformations sera décrite plus loin.

6. Variables

METAPOST autorise des noms de variable composés tels que `x.a`, `x2r`, `y2r` et `z2r` où `z2r` signifie `(x2r,y2r)` et `z.a` signifie `(x.a,y.a)`. En fait, il existe une classe entière de suffixes tels que `z`<suffixe> est équivalent à

(x <suffixe>, y <suffixe>)

Puisqu'un <suffixe> est composé de *tokens* il est préférable de commencer avec quelques remarques sur les *tokens*.

6.1. *Tokens*

Un fichier source METAPOST est traité comme une séquence de nombres, de constantes chaînes de caractères et de *tokens* symboliques. Un nombre consiste en une séquence de chiffres pouvant contenir un point décimal. Techniquement, le signe moins, devant un nombre négatif, est un *token* à part entière. Puisque METAPOST utilise l'arithmétique à point fixe, il ne comprend pas la notation exponentielle telle que `6.02E23`. METAPOST interpréterait ceci comme le nombre `6.02` suivi par le *token* symbolique `E`, suivi enfin par le nombre `23`.

Tout ce qui se trouve entre entre une paire de guillemets (double-quotes) est une constante chaîne de caractères. Une chaîne ne peut pas commencer sur une ligne pour se terminer sur la ligne suivante. De même, une chaîne ne peut pas contenir des guillemets ou tout caractère autre que les caractères `ascii` imprimables.

Dans une ligne du fichier source, tout élément autre que les nombres et les constantes chaînes est considéré comme un *token* symbolique. Un *token* symbolique est une séquence d'un ou plusieurs caractères de même catégories, où les caractères sont « de même catégories » s'ils apparaissent dans la même ligne du tableau 1.

Ainsi `A_alpha` et `+-+` sont des *tokens* symboliques mais `!=` est interprété comme deux *tokens* et `x34` est un *token* symbolique suivi par un nombre. Puisque les crochets `[` et `]` sont listés sur une ligne chacun, les seuls *tokens* symboliques les mettant en jeu sont `[`, `[[`, `[[[`, etc. et `]`, `]]`, `]]]` etc.

Quelques caractères ne sont pas listés dans le tableau 1 parce qu'ils nécessitent un traitement particulier. Les quatre caractères `,;()` sont des « solitaires » : chaque virgule, point-virgule ou parenthèse est un *token* à part entière même s'ils apparaissent consécutivement. Ainsi, `(())` constitue quatre *tokens*, et non un ou deux. Le signe pourcentage `%` est très spécial parce qu'il introduit des commentaires. Le signe `%` et tout ce qui suit jusqu'à la fin de la ligne sera ignoré.

Un autre caractère spécial est le point (`.`). Deux ou plusieurs points forment ensemble un *token* symbolique, mais un seul point est ignoré. Un point précédé ou suivi par des chiffres est une partie du nombre. Ainsi `..` ou `...` sont des *tokens* symboliques alors que `a.b` ne représentent que deux *tokens* `a` et `b`. Il est conventionnel d'utiliser des points pour séparer des *tokens* de cette façon lorsqu'on veut nommer une variable qui est plus longue qu'un *token*.

TAB. 1 – Classes de caractères pour les *tokens*

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ_abcdefghijklmnopqrstuvwxyz
                        :<=>|
                         #&@$
                          /*\
                           !?
                           ´`
                           ^~
                           {}
                            [
                            ]
```

6.2. Déclarations de variables

Un nom de variable est un *token* symbolique ou une séquence de *tokens* symboliques. La plupart des *tokens* symboliques constituent des noms de variables légitimes mais tout ce qui possède un sens prédéfini tel que `draw`, `+` ou `..` est interdit. Les noms de variables ne peuvent pas utiliser les noms de macros ou de primitives METAPOST. Cette restriction mineure permet une variété extraordinaire de noms de variables : `alpha`, ==>, `@&#$&` et $\sim\sim$ sont tous des noms de variables légitimes. De tels *tokens* symboliques sans signification particulière sont appelés des *tags*.

Un nom de variable peut être une séquence de *tags* telle que `f.bot` ou `f.top`. L'idée est de profiter de quelques fonctionnalités des *records* du Pascal ou des *structs* du langage C. Il est aussi possible de simuler des tableaux *(arrays)* en utilisant des noms de variable contenant des nombres aussi bien que des *tokens* symboliques. Par exemple, le nom de variable `x2r` est constitué par le *tag* `x`, le nombre `2` et le *tag* `r`. On peut avoir des variables nommées `x3r` et même `x3r.14r`. Ces variables peuvent être traitées comme un tableau par des constructions telles que `x[i]r` où `i` a une valeur numérique appropriée. La syntaxe générale pour les noms de variables se trouve dans la figure 16.

<variable>→ <*tag*><suffixe>
<suffixe>→ <vide> | <suffixe><indice> | <suffixe><*tag*>
<indice>→ <nombre> |[<expression numérique>]

FIG. 16 – Syntaxe des noms de variable

Des variables telles que `x2` et `y2` acceptent par défaut des valeurs numériques, on peut ainsi utiliser le fait que `z`<suffixe> est une abréviation pour

$$(x \text{ <suffixe>}, y \text{ <suffixe>})$$

pour créer une variable de type couple de valeurs si nécessaire.

La macro `beginfig` efface les valeurs pré-existantes de variables qui commencent avec les *tags* `x` ou `y` de sorte que les blocs `beginfig... endfig` n'interfèrent pas les uns avec les autres lorsque ce schéma de nom est utilisé. En d'autres termes, les variables qui commencent avec `x`, `y`, `z` sont locales à la figure à laquelle elles se rapportent. Des mécanismes généraux pour fabriquer des variables locales seront exposés dans le paragraphe 9.1

La déclaration

```
pair pp, a.b;
```

rend les couples `pp` et `a.b` inconnus. Une telle déclaration n'est pas strictement locale puisque `pp` et `a.b` ne sont pas automatiquement restaurés à leurs valeurs initiales à la

fin de la figure courante *(current picture)*. Naturellement, ils redeviennent des couples indéterminés si la déclaration est faite une nouvelle fois.

Les déclarations fonctionnent de la même façon pour n'importe lequel des huit autres types : numérique, chemin, transformation, couleur, chaîne, booléen, figure et stylo. La seule restriction est qu'on ne peut pas donner explicitement des indices numériques dans une déclaration de variable. Il ne faut pas donner la déclaration, illégale,

```
numeric q1, q2, q3;
```

mais utiliser à la place, le symbole d'indice générique `[]` pour déclarer un tableau entièrement :

```
numeric q[];
```

On peut aussi définir des tableaux « multidimensionnels » . Après la déclaration

```
path p[]q[], pq[][];
```

`p2q3` et `pq1.4 5` sont tous les deux des chemins.

Des variables internes telles que `tracingonline` ne peuvent pas être déclarées normalement de cette façon. Toutes les variables internes dont il est question dans ce document sont prédéfinies et ne doivent pas être déclarées du tout. Cependant il existe un moyen pour déclarer qu'une variable doit se comporter comme une variable interne nouvellement créée. La déclaration est `newinternal` suivie d'une liste de *tokens* symboliques.

Par exemple, la commande

```
newinternal a, b, c;
```

oblige `a`, `b` et `c` à se comporter comme des variables internes. De telles variables doivent toujours avoir des valeurs numériques connues et ces valeurs ne peuvent être modifiées que par l'opérateur d'assignation `:=`. Les variables internes sont initialisées à zéro sauf celles qui sont initialisées par le package de macro Plain (les macros Plain sont normalement préchargées automatiquement comme l'explique le chapitre 1).

7. Intégration de textes et de graphiques

METAPOST dispose de nombreux moyens pour inclure des labels (ou étiquettes) et d'autres textes dans les figures qu'il crée. Le moyen le plus simple pour cela est d'utiliser l'instruction `label`

`label`<suffixe d'étiquette>(<expression chaîne ou dessin>, <expression paire>);

L'<expression chaîne ou dessin> indique l'étiquette et l'<expression paire> dit où la mettre. Le <suffixe d'étiquette> peut être <vide>, dans ce cas l'étiquette sera juste centrée au point choisi. Si l'on veut, on peut décaler légèrement l'étiquette pour éviter des recouvrements. Ceci est illustré dans la figure 17 où l'étiquette `"a"` est placée au dessus du milieu de la ligne à laquelle elle se réfère et le `"b"` est à gauche du milieu de la ligne verticale. Ceci est obtenu en utilisant `label.top` pour `"a"` et `label.lft` pour `"b"`. La commande <suffixe d'étiquette> spécifie la position de l'étiquette relativement au point choisi. Le jeu complet de possibilités est

<suffixe d'étiquette>→ <vide> |`lft`|`rt`|`top`|`bot`|`ulft`|`urt`|`llft`| `lrt`

où `lft` et `rt` signifient left (gauche) et right (droite) et `llft`, `ulft`, etc. signifient lower left (gauche bas), upper left (gauche haut), etc. La distance réelle avec laquelle l'étiquette est décalée est déterminée par la variable interne `labeloffset`.

```
beginfig(17);
a=.7in; b=.5in;
z0=(0,0);
z1=-z3=(a,0);
z2=-z4=(0,b);
draw
z1..z2..z3..z4..cycle;
draw z1 -- z0 -- z2;
label.top("a",
.5[z0,z1]);
label.lft("b",
.5[z0,z2]);
dotlabel.bot("(0,0)",
z0);
endfig;
```

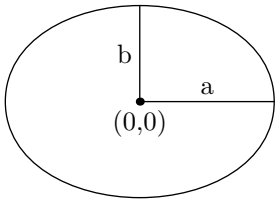


FIG. 17 – Code METAPOST et son résultat

La figure 17 illustre la commande `dotlabel`. Elle est tout à fait identique à la commande `label` à la différence qu'elle ajoute un point au point de coordonnées indiqué. Par exemple

```
dotlabel.bot("(0,0)", z0)
```

place un point à `z0` puis met l'étiquette « (0,0) » juste en-dessous du point.

Une autre possibilité est la macro `thelabel`. Cette macro a la même syntaxe que les commandes `label` et `dotlabel` mis à part qu'elle renvoie l'étiquette comme <dessin primaire> au lieu de la dessiner réellement. Ainsi

```
label.bot("(0,0)", z0)
```

est équivalent à

```
draw thelabel.bot("(0,0)", z0)
```

Pour des applications simples on peut juste se servir des commandes `label` et `dotlabel`. En fait, on a la possibilité d'utiliser une forme raccourcie de la commande `dotlabel` qui évite beaucoup de frappes lorsque les points sont nombreux : `z0`, `z1`, `z.a`, `z.b`, etc. Dans ce cas on peut utiliser les suffixes comme étiquettes. La commande

```
dotlabels.rt(0, 1, a);
```

est équivalente à

```
dotlabel.rt("0",z0);dotlabel.rt("1",z1);dotlabel.rt("a",z.a);
```

Ainsi l'argument de `dotlabels` est une liste de suffixes pour lesquels les variables `z` sont connues et le <suffixe d'étiquette> donné avec `dotlabels` est utilisé pour positionner toutes les étiquettes.

Il existe aussi une commande `labels` qui est analogue à `dotlabels` mais son utilisation est à éviter parce qu'elle présente des problèmes de compatibilité avec METAFONT. Certaines versions de Plain préchargé définissent `labels` comme synonyme de `dotlabels`.

Pour les commandes d'étiquettes telles que `label` et `dotlabel` qui utilisent une chaîne de caractères pour texte, la chaîne sera composée dans la fonte par défaut qui est déterminée par la variable chaîne `defaultfont`. La valeur initiale de `defaultfont` est généralement `"cmr10"`, mais elle peut être changée par une affectation du type

```
defaultfont := "Times-Roman"
```

Il existe aussi une quantité numérique appelée `defaultscale` qui détermine la taille des caractères. Quand `defaultscale` est égale à 1, on obtient la « taille normale » qui est généralement de 10 points, mais cette taille peut changer. Par exemple

```
defaultscale := 1.2
```

augmente la taille de 20 %. Si l'on ne connaît pas la taille normale et que l'on veut être sûr que le texte soit d'une taille certaine, disons 12 points par exemple, on peut utiliser l'opérateur `fontsize` pour déterminer la taille normale :

```
defaultscale := 12pt/fontsize defaultfont;
```

Quand on change `defaultfont`, le nouveau nom de fonte doit être quelque chose que TEX peut interpréter puisque METAPOST donne la hauteur et la largeur en lisant le fichier `tfm` (ceci est expliqué dans le *TEXbook* [5]). Il serait possible d'utiliser des fontes POSTSCRIPT, mais leur nom dépend du système. Certains systèmes peuvent utiliser `rptmr` ou `ps-times-roman` au lieu de `Times-Roman`. Une fonte TEX, telle que `cmr10`, est un peu dangereuse parce qu'elle ne possède pas de caractère espace ni certains symboles ASCII. De plus, METAPOST n'utilise pas les informations de ligature qui sont attachées aux fontes TEX.

7.1. Composer les étiquettes *(labels)*

TEX peut être utilisé pour mettre en forme des étiquettes complexes. Si l'on écrit

`btex`<commandes de mise en forme>`etex`

dans un fichier source METAPOST, le paramètre <commandes de mise en forme> est traité par TEX et traduit en une expression de type figure (en réalité une <figure primaire>) qui peut être utilisée dans une déclaration `label` ou `dotlabel`. Tout espace après `btex` ou avant `etex` est ignoré. Par exemple, la déclaration

```
label.lrt(btex $ \sqrt x $ etex, (3,sqrt 3)*u)
```

dans la figure 18 place l'étiquette $\sqrt{x}$ sous et à droite du point `(3,sqrt 3)*u`.

```
beginfig(18);
numeric u;
u = 1cm;
draw (0,2u) -- (0,0) -- (4u,0);
pickup pencircle scaled 1pt;
draw (0,0){up}
   for i=1 upto 8 :
..(i/2,sqrt(i/2))*u endfor;
label.lrt(btex $ \sqrt x$ etex,
(3,sqrt 3)*u);
label.bot(btex $x$ etex, (2u,0));
label.lft(btex $y$ etex, (0,u));
endfig;
```

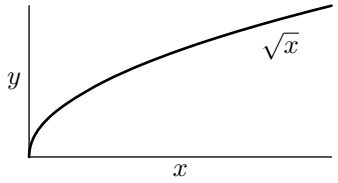


FIG. 18 – Code METAPOST et la sortie correspondante

```
beginfig(19);
numeric ux, uy;
120ux=1.2in; 4uy=2.4in;
draw (0,4uy) -- (0,0) --
(120ux,0);
pickup pencircle scaled 1pt;
draw (0,uy){right}
for ix=1 upto 8 :
..(15ix*ux, uy*2/(1+cosd
15ix))
endfor;
label.bot(btex axe $x$ etex,
(60ux,0));
label.lft(btex axe $y$ etex
rotated 90, (0,2uy));
label.lft(btex
$\displaystyle
y={2\over1+\cos x}$ etex,
(120ux, 4uy));
endfig;
```

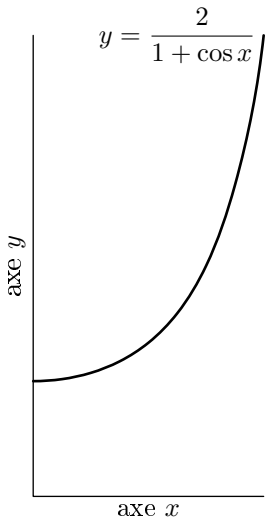


FIG. 19 – Code METAPOST et sa sortie correspondante

La figure 19 illustre quelques unes des choses les plus compliquées que l'on peut faire avec les étiquettes. Puisque le résultat de `btex... etex` est une figure *(picture)*, elle peut être manipulée comme une figure. En particulier, il est possible d'appliquer des transformations. Jusqu'ici, la syntaxe pour ceci n'a pas été mentionnée, mais une <figure secondaire> peut être

<figure secondaire> `rotated` <numérique primaire>

Ceci est utilisé dans la figure 19, pour tourner l'étiquette « *axe y* » afin qu'elle soit disposée verticalement.

Une autre complication dans la figure 19 est l'utilisation et l'affichage de l'équation

$$y = \frac{2}{1+\cos x}$$

comme label. Il serait plus naturel de le coder comme

```
$$y = {2\over 1+\cos x}$$
```

mais cela ne marcherait pas parce que TEX compose les étiquettes en mode horizontal.

Le processeur METAPOST saute les blocs `btex... etex` et laisse la traduction de ce dernier à un préprocesseur pour le transcrire en commandes METAPOST de bas niveau. Si le fichier principal est `fig.mp`, les données traduites par TEX sont placées dans un fichier appelé `fig.mpx`. Ceci se fait normalement de façon silencieuse sans aucune intervention de l'utilisateur mais ce processus peut échouer si un des blocs `btex... etex` contient une commande TEX erronée. Le fichier TEX erroné est sauvé dans le fichier `mpxerr.tex` et les messages d'erreurs apparaîssent dans le fichier `mpxerr.log`.

Les définitions de macro TEX ou toute commande TEX auxiliaire peuvent être entourées dans un bloc `verbatimtex... etex`. La différence entre `btex` et `verbatimtex` est que la première produit une expression image alors que la dernière ajoute simplement des informations pour que TEX compose. Par exemple, si l'on veut que TEX compose des étiquettes en utilisant des macros définies dans `mymac.tex`, le fichier source METAPOST doit ressembler à quelque chose comme

```
verbatimtex \input mymac etex
beginfig(1);
   ⋮
label(btex<matériel TEX utilisant mymac.tex>etex,  <coordonnées>);
   ⋮
```

Sur des systèmes Unix, on peut utiliser une variable d'environnement pour spécifier que les blocs `btex... etex` et `verbatimtex... etex` sont en troff au lieu d'être en TEX. Lorsque cette option est utilisée, une bonne idée est de commencer le fichier METAPOST avec la déclaration `prologues :=1`. Donner une valeur positive à cette variable interne permet de mettre le fichier de sortie au format « structured POSTSCRIPT » produit sur l'hypothèse que le texte utilise des fontes POSTSCRIPT. Cela augmente la portabilité des sorties POSTSCRIPT mais la contrepartie est importante : cela ne marche généralement pas lorsqu'on utilise des fontes TEX puisque les programmes qui traduisent du TEX en POSTSCRIPT ont besoin de renseignements spéciaux pour les fontes TEX inclues dans des figures, or les règles de structuration standard PS ne permettent pas cela. Les détails sur la manière d'inclure des figures POSTSCRIPT dans un document produit par TEX ou en troff, dépendent du système. Généralement, ils peuvent être trouvés dans les pages du manuel ou dans d'autres documentations en ligne. Un fichier appelé `dvips.tex` est distribué électroniquement avec le processeur `dvips`.

7.2. Opérateur `infont`

Sans tenir compte de savoir si on utilise TEX ou troff, tout le travail réel d'insertion de textes dans les figures est fait par une primitive METAPOST nommée `infont`. C'est un <opérateur binaire primaire> qui requiert une <chaîne secondaire> comme argument à gauche et une <chaîne primaire> comme argument à droite. L'argument gauche est un texte et l'argument droit est un nom de fonte. Le résultat de l'opération est une <image secondaire> qui peut ensuite être transformée de multiples façons. Une possibilité est l'élargissement par un facteur donné par la syntaxe :

<image secondaire>`scaled`<numérique primaire>

Ainsi `label("texte",z0)` est équivalent à

```
label("texte" infont defaultfont scaled defaultscale, z0)
```

S'il n'est pas pratique d'utiliser une constante chaîne de caractères comme argument gauche de `infont`, on peut utiliser

`char`<numérique primaire>

pour sélectionner un caractère en se basant sur sa position dans la fonte. Ainsi

```
char(n+64) infont "Times-Roman"
```

est une image contenant le caractère `n+64` de la fonte Times-Roman.

7.3. Mesurer du texte

METAPOST permet d'atteindre facilement les dimensions physiques des figures créées par l'opérateur `infont`. Les opérateurs unaires `llcorner`, `lrcorner`, `urcorner`, `ulcorner` et `center` s'appliquent à une <image primaire> et renvoient les coins de sa *bounding box* , comme l'illustre la figure 20. L'opérateur `center` accepte également des opérandes de type <chemin primaire> et <stylo primaire>. Dans les versions 0.30 et supérieures de METAPOST, `llcorner`, `lrcorner`, etc. acceptent les trois types d'arguments.

FIG. 20 – Une *bounding box* et ses points de coin

Les restrictions sur le type des arguments pour les opérateurs de coins ne sont pas très importantes parce que leur propos principal est de permettre aux déclarations `label` et `dotlabel` de centrer correctement leur texte. La macro prédéfinie

`bbox`<image primaire>

trouve un chemin rectangulaire qui représente la « bounding box » d'une figure donnée.

Si `p` est une figure, `bbox p` équivaut à

```
(llcorner p -- lrcorner p -- urcorner p -- ulcorner p -- cycle)
```

sauf qu'elle permet d'ajouter un petit espace supplémentaire autour de `p`, espace qui est spécifié par la variable interne `bboxmargin`.

Il faut noter que METAPOST calcule la *bounding box* d'une image `btex .... etex` exactement de la même façon que le ferait TEX. Ceci est assez naturel mais cela entraîne certaines implications en raison du fait que TEX possède des possibilités telles que `\strut` ou `\rlap` qui permettent à l'utilisateur de mentir sur les dimensions d'une boîte.

Quand des commandes TEX concernant les dimensions d'une boîte sont traduites en code bas niveau METAPOST, une instruction `setbounds`

`setbounds`<variable image> `to` <expression chemin>

permet à la <variable image> de se comporter comme si sa *bounding box* était le chemin donné. Pour obtenir le véritable *bounding box* d'une telle figure, il faut affecter une valeur positive à la variable interne `truecorner` [5] ; donc

```
show urcorner btex $\bullet$\rlap{ A} etex
```

produit « >>`(4.9813,6.8078)` » alors que

```
truecorners :=1 show urcorner btex $\bullet$\rlap{ A} etex
```

produit « >>`(15.7742,6.8078)` ».

8. Graphiques avancés

Tous les exemples dans les sections précédentes sont tracés avec des lignes simples et avec des ajouts d'étiquettes. Cette section décrit comment remplir des tracés et les outils pour produire des lignes moins simples.

Le remplissage est obtenu avec l'instruction `fill`. Dans sa forme élémentaire, l'instruction `fill` requiert une <expression chemin> qui donne la limite de la région à remplir. Dans la syntaxe

`fill`<expression chemin>

l'argument doit être un chemin fermé, c'est-à-dire un chemin qui décrit une courbe fermée par la notation `..cycle` ou `-- cycle`. Par exemple, la déclaration `fill` de la figure 21 construit un chemin fermé par extension du chemin `p` approximativement semi-circulaire. Ce chemin est orienté dans le sens trigonométrique ou dans le sens inverse des aiguilles d'une montre, mais cela importe peu parce que la déclaration `fill` utilise la règle POSTSCRIPT de nombre de tours non nul [1].

L'instruction `fill` générale

`fill`<expression chemin>`withcolor`<expression couleur>

[5] Les caractéristiques `setbounds` et `truecorner` n'existent que dans les versions 0.30 et ultérieures de METAPOST

spécifie un remplissage gris ou en couleur (à condition de posséder une imprimante couleur).

```
beginfig(21);
path p;
p = (-1cm,0)..(0,-1cm)..(1cm,0);
fill p{up} .. (0,0){-1,-2} ..
{up}cycle;
draw p .. (0,1cm) .. cycle;
endfig;
```

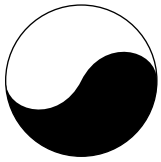


FIG. 21 – Code METAPOST et sa sortie correspondante

```
beginfig(22);
path a, b, aa, ab;
a = fullcircle scaled 2cm;
b = a shifted (0,1cm);
aa = halfcircle scaled 2cm;
ab = buildcycle(aa, b);
picture pa, pb;
pa = thelabel(btex $A$ etex, (0,-
.5cm));
pb = thelabel(btex $B$ etex,
(0,1.5cm));
fill a withcolor .7white;
fill b withcolor .7white;
fill ab withcolor .4white;
unfill bbox pa;
draw pa;
unfill bbox pb;
draw pb;
label.lft(btex $U$ etex, (-
1cm,.5cm));
draw bbox currentpicture;
endfig;
```

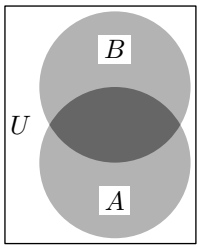


FIG. 22 – Code METAPOST et la sortie correspondante

La figure 22 illustre plusieurs applications de la commande `fill` pour remplir des aires avec différents gris. Les chemins concernés sont les cercles `a` et `b` se coupant et le chemin `ab` qui délimite l'intersection des deux cercles. Les cercles `a` et `b` sont dérivés d'un chemin prédéfini `fullcircle` qui calcule approximativement un cercle de diamètre unité centré sur l'origine. Il existe également un chemin prédéfini

`halfcircle` qui est la partie de `fullcircle` située au dessus de l'axe des x. Le chemin `ab` est construit par la macro prédéfinie `buildcycle` qui sera étudiée bientôt.

Remplir les cercles `a` et `b` avec un gris clair `.7white` provoque un remplissage double ou une superposition de couleurs dans la région de recouvrement des deux cercles. La règle est que chaque déclaration `fill` assigne la couleur choisie à tous les points de la région concernée, annulant tout ce qui existait auparavant, incluant lignes et textes aussi bien que les domaines remplis. Ainsi est-il important d'appliquer les commandes `fill` dans le bon ordre. Dans l'exemple précédent, la région de recouvrement reçoit deux fois la même couleur, laissant le gris clair après les deux premières commandes `fill`. La troisième commande `fill` assigne la couleur plus foncée `.4white` dans la zone de recouvrement.

À ce point, les cercles et la zone de recouvrement ont leur couleur finale mais il n'y a pas de découpe pour les étiquettes. Les découpes sont obtenues par la commande `unfill` qui efface effectivement les régions limitées par `bbox pa` et `bbox pb`. Plus précisément, la commande `unfill` est un raccourci pour un remplissage `withcolor background`, où `background` est normalement égal à `white` (blanc) conformément à une impression sur du papier blanc. Si cela est nécessaire, on peut toujours assigner une nouvelle couleur à `background`.

Les étiquettes doivent être stockées dans des figures *(picture)* `pa` et `pb` pour permettre la mesure de leur *bounding box* avant de les tracer réellement. La macro `thelabel` crée de telles figures et les décale à la position prévue pour l'impression. L'utilisation des figures résultantes dans une déclaration `draw` de la forme

`draw`<expression figure>

les ajoute à la `currentpicture` (ou *figure courante*) de telle sorte qu'elles se superposent sur une portion de ce qui a déjà été dessiné. Dans la figure 22, seuls les rectangles blancs produits par `unfill` sont superposés.

8.1. Construction des chemins fermés ou cycles

La commande `buildcycle` construit des chemins pour être utilisés avec les macros `fill` ou `unfill`. Soient au moins deux chemins tels que `aa` et `b` (figure 23), la macro `buildcycle` permet de les joindre de façon à former un chemin cyclique ou fermé. Dans le cas présent, le chemin `aa` est un demi-cercle qui commence juste à droite de l'intersection 1 avec le chemin `b`, puis passe à travers `b` et se termine à l'extérieur du cercle juste à gauche comme le montre la figure 23a.

La figure 23b montre comment `buildcycle` forme un chemin fermé à partir des morceaux de chemins `aa` et `b`. La macro `buildcycle` détecte les deux intersections

étiquetées 1 et 2 dans la figure 23b. Ensuite elle construit le chemin, en gras sur la figure, en parcourant le chemin `aa` depuis l'intersection 1 jusqu'à l'intersection 2 et ensuite en suivant, dans le sens trigonométrique, le chemin `b` jusqu'à l'intersection 1. Il se trouve que la commande `buildcycle(aa,b)` aurait produit le même résultat, mais le raisonnement derrière cette commande est un peu difficile à comprendre.

FIG. 23 – (a) Le chemin semi-circulaire `aa` et le chemin `b` en pointillé ; (b) chemins `aa` et `b` et les parties sélectionnées par `buildcycle`, en ligne épaisse

Il est facile d'utiliser la macro `buildcycle` dans des situations telles que celle de la figure 24 dans laquelle il y a plus de deux chemins et chaque couple de chemins consécutifs a une intersection unique. Par exemple, la ligne `q0.5` et la courbe `p2` se coupent uniquement au point P et la courbe `p2` et la ligne `q1.5` se coupent uniquement au point Q. En fait, chacun des points P, Q, R et S est un point d'intersection unique. Le résultat de

```
buildcycle(q0.5, p2, q1.5, p4)
```

considère `q0.5` entre S et P, puis `p2` entre P et Q, puis `q1.5` entre Q et R et finalement `p4` entre R et S. Un examen du code source METAPOST(figure 24) révèle qu'il faut revenir en arrière le long de `p2` pour aller de P à Q. Ceci fonctionne très bien tant que les points d'intersection sont uniques. Cela peut donner des résultats inattendus quand les couples de chemins possèdent plusieurs points d'intersection.

La règle générale pour la macro `buildcycle` est que

$$\texttt{buildcycle(}p_1, p_2, p_3, \ldots, p_k\texttt{)}$$

choisit le point entre chaque p_i et p_{i+1} le plus tardif possible sur p_i et le plus récent possible sur p_{i+1}. Il n'y a pas de règles simples pour résoudre les conflits entre ces deux objectifs. On doit éviter les cas où un point d'intersection intervient plus loin sur p_i et un autre point d'intersection intervient plus tôt sur p_{i+1}.

La préférence pour les intersections le plus tard possible sur p_i et le plus tôt sur p_{i+1} conduit à la résolution de cette ambiguïté en faveur de l'exploration plus en aval de sous-chemins. Pour des chemins cycliques tels que le chemin `b` de la figure 23,

```
beginfig(24);
h=2in; w=2.7in;
path p[], q[], pp;
for i=2 upto 4 : ii :=i**2;
p[i] = (w/ii,h){1,-ii}...(w/i,h/i)...(w,h/ii){ii,-1};
endfor
q0.5 = (0,0) -- (w,0.5h);
q1.5 = (0,0) -- (w/1.5,h);
pp = buildcycle(q0.5, p2, q1.5, p4);
fill pp withcolor .7white;
z0=center pp;
picture lab;
lab=thelabel(btex $f>0$ etex, z0);
unfill bbox lab; draw lab; draw q0.5; draw p2; draw q1.5;
draw p4;
dotlabel.top(btex $P$ etex, p2 intersectionpoint q0.5);
dotlabel.rt(btex $Q$ etex, p2 intersectionpoint q1.5);
dotlabel.lft(btex $R$ etex, p4 intersectionpoint q1.5);
dotlabel.bot(btex $S$ etex, p4 intersectionpoint q0.5);
endfig;
```

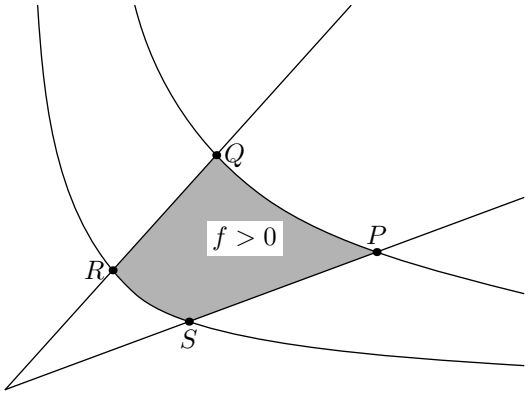


FIG. 24 – Code source METAPOST et la sortie associée

les notions « récente » et « tardive » sont relatives à un point de départ et de fin qui se trouve là où l'on revient lorsqu'on indique « `..cycle` » . pour le chemin `b`, cela correspond au point le plus à droite sur le cercle.

Une façon plus directe de s'occuper de l'intersection des chemins passe par l'<opérateur binaire secondaire> `intersectionpoint` qui trouve les points P, Q,

R et S dans la figure 24. Cette macro trouve un point à l'intersection de deux chemins donnés. S'il existe plus de deux points d'intersection, elle en choisit juste un, s'il n'en existe pas, elle renvoie un message d'erreur.

8.2. Gestion paramétrique des chemins

La macro `intersectionpoint` est basée sur une primitive appelée `intersectiontimes`. Cet <opérateur binaire secondaire> est une des nombreuses opérations qui s'intéressent aux chemins définis paramétriquement. Il localise une intersection entre deux chemins en donnant le paramètre t ou « temps » (time dans la version originale) sur chaque chemin. Ceci se réfère au schéma de paramétrisation du chapitre 3 qui décrit les chemins comme des assemblages de cubiques $(X(t),\ Y(t))$ où t parcourt les valeurs depuis zéro jusqu'au nombre de segments de courbe. En d'autres termes, quand un chemin est spécifié comme passant par un série de points, alors $t = 0$ au premier point, $t = 1$ au suivant, $t = 2$ au suivant, etc. Le résultat de

```
a intersectiontimes b
```

est (-1, -1) s'il n'y a pas de point d'intersection, sinon on obtient un couple (t_a, t_b), où t_a est le « temps » sur le chemin `a` quand il rencontre le chemin `b`, et t_b est le « temps » sur le chemin `b`.

Par exemple, supposons que le chemin `a` correspond à la courbe fine sur la figure 25 et que le chemin `b` correspond à la courbe plus épaisse. Si les étiquettes indiquent les valeurs « temps » sur les chemins, le couple de valeurs « temps » calculées par

```
a intersectiontimes b
```

doit être l'un des couples suivants :

(0.25,1.77), (0.75, 1.40) ou (2.58, 0.24)

suivant le point d'intersection qu'aura choisi l'interpréteur de METAPOST. Les règles exactes pour choisir parmi plusieurs points d'intersection sont un peu compliquées, mais il se trouve qu'on obtiendra le couple de valeurs (0.25,1.77) dans cet exemple. Les valeurs préférées sont plutôt les plus petites que les plus grandes. Ainsi (t_a, t_b) sera préféré à (t'_a, t'_b) chaque fois que $t'_a < t_a$ et $t'_b < t_b$. Lorsqu'aucune possibilité ne minimise à la fois les composantes t_a et t_b, la composante t_a tend à avoir la priorité, mais les règles deviennent plus compliquées lorsqu'il n'y a pas d'entier entre t_a et t'_a. (Pour plus de détails, voir le *METAFONTbook* [4])

L'opérateur `intersectiontimes` est plus souple que `intersectionpoint` parce qu'un certain nombre de choses peuvent être faites avec les valeurs « temps » sur un chemin. Une des plus importantes est justement de demander « où est le chemin `p` pour le paramètre `t` ? » La construction

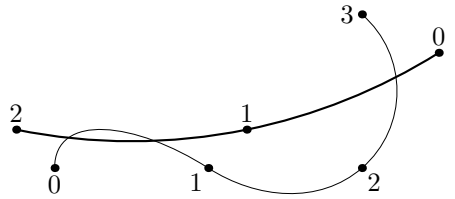


FIG. 25 – Deux chemins se coupant avec les valeurs de t sur chaque courbes

`point`<expression numérique> `of` <chemin primaire>

répond à cette question. Si l'<expression numérique> est inférieure à zéro ou supérieure à la valeur t assignée au dernier point du chemin, la construction `point of` produit un point final du chemin. Désormais, il sera courant d'utiliser la constante prédéfinie `infinity` (égale à 4095.99998) comme l'<expression numérique> dans une construction `point of` quand on s'intéresse à l'extrémité d'un chemin.

De telles valeurs « infinies » de t ne fonctionnent pas pour des chemins cycliques, puisque des valeurs de t en dehors de l'intervalle normal peuvent être manipulées par l'arithmétique modulaire dans ce cas ; c'est-à-dire pour un chemin cyclique `p` passant par les points z_0, z_1, z_2, ... , z_{n-1}, t varie normalement dans l'intervalle $[0, n]$, mais

```
point t of p
```

peut être calculé pour tout t en réduisant d'abord t modulo n.

Si le modulo n n'est pas réellement accessible,

`length`<chemin primaire>

donne la valeur entière de la limite supérieure de l'intervalle normal du paramètre temps pour le chemin spécifié.

METAPOST utilise les mêmes correspondances entre les valeurs de t et les points sur la courbe pour évaluer l'opérateur `subpath`. La syntaxe pour cet opérateur est

`subpath`<expression paire> `of`<chemin primaire>

Si la valeur de l'<expression paire> est (t_1, t_2) et si le <chemin primaire> est p, le résultat est un chemin qui suit p depuis le `point` t_1 `of` p jusqu'au `point` t_2 `of` p. Si $t_2 < t_1$ le sous-chemin suit p en sens opposé.

Une opération importante sur l'opérateur `subpath` est l'<opérateur binaire tertiaire> `cutbefore`. Pour trouver l'intersection des chemins p_1 et p_2,

p_1 `cutbefore` p_2

est équivalent à

`subpath (xpart(`p_1 `intersectiontimes` p_2`), length` p_1`) of` p_1

excepté le fait qu'il met la portion de p_1, qu'il coupe, dans la variable de type chemin `cuttings`. En d'autres termes, `cutbefore` renvoie le premier argument avec la partie précédant la coupure. Avec de multiples intersections, il essaye de supprimer la partie la plus petite possible. Si les chemins ne se croisent pas, `cutbefore` renvoie son premier argument.

Il existe un opérateur analogue appelé `cutafter` qui fonctionne par application de `cutbefore` avec une inversion de t le long du premier argument. Ainsi

p_1 `cutafter` p_2

essaye de couper la partie de p_1 après sa dernière intersection avec p_2.

Un autre opérateur

`direction`<expression numérique>`of`<chemin primaire>

renvoie un vecteur dans la direction du <chemin primaire>. Ceci est défini pour toute valeur de t de façon analogue à la construction `point of`. Le vecteur direction résultant possède l'orientation correcte et une amplitude quelque peu arbitraire. La combinaison de `point of` et `direction of` produit l'équation pour une ligne tangente, comme l'illustre la figure 26.

Si l'on connaît la pente et que l'on veut trouver un point sur une courbe dont la tangente a cette pente, l'opérateur `directiontime` inverse l'opérateur `direction of`.

Soient un vecteur et un chemin, la commande

`directiontime`<expression paire> `of`<chemin primaire>

renvoie une valeur numérique qui donne la première valeur de t quand le chemin a la direction indiquée ; s'il n'existe pas une telle valeur de t, le résultat est -1. Par exemple, si `a` est le chemin dessiné en trait fin dans la figure 25, `directiontime (1,1) of a` renvoie 0.2084.

```
beginfig(26);
numeric scf, #, t[];
3.2scf = 2.4in;
path fun;
# = .1; % Keep the function single-valued
fun = ((0,-1#) .. (1,.5#){right} .. (1.9,.2#){right} ..
{curl 0.1}(3.2,2#)) yscaled(1/#) scaled scf;
x1 = 2.5scf;
for i=1 upto 2 :
   (t[i],whatever) =
   fun intersectiontimes ((x[i],-infinity) --
(x[i],infinity));
   z[i] = point t[i] of fun;
   z[i]-(x[i+1],0) = whatever*direction t[i] of fun;
   draw (x[i],0) -- z[i] -- (x[i+1],0);
   fill fullcircle scaled 3bp shifted z[i];
endfor
label.bot(btex $x_1$ etex, (x1,0));
label.bot(btex $x_2$ etex, (x2,0));
label.bot(btex $x_3$ etex, (x3,0));
draw (0,0) -- (3.2scf,0);
pickup pencircle scaled 1pt;
draw fun;
endfig;
```

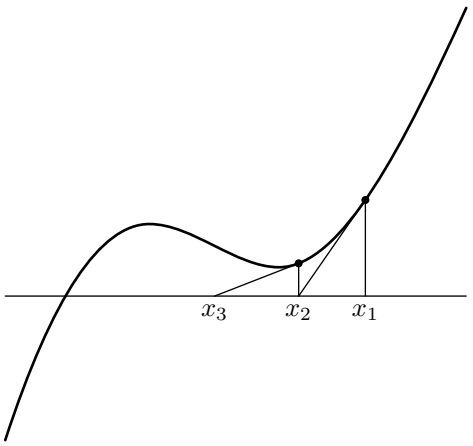


FIG. 26 – Code METAPOST et figure résultante

Il existe aussi une macro prédéfinie

`directionpoint`<expression paire>`of`<chemin primaire>

qui trouve le premier point d'un chemin donné pour lequel une direction est donnée. La macro `directionpoint` produit un message d'erreur si la direction n'est pas trouvée sur le chemin.

Les opérateurs `arclength` et `arctime of` sont reliés au paramètre t d'un chemin et sont rattachés au concept plus familier de longueur d'un arc[6]. L'expression

`arclength`<primaire chemin>

donne la longueur d'un chemin. Si `p` est un chemin et si `a` est un nombre entre 0 et `arclength p`

```
arctime a of p
```

donne la valeur du paramètre t telle que `arclength subpath (0,t) of p = a`

8.3. Transformations affines

Le chemin `fun` de la figure 26 a été construit à l'aide de la commande

```
(0,-.1)..(1,.05){right}..(1.9,.02){right}..{curl 0.1}(3.2,.2)
```

et ensuite les opérateurs `yscaled` et `scaled` sont utilisés pour ajuster la forme et la taille du chemin. Comme le nom le suggère, une expression mettant en jeu « `yscaled 10` » multiplie la coordonnée y par 10 de telle sorte qu'à chaque point $(x,\ y)$ du chemin original est associé le point $(x,\ 10y)$ du chemin transformé.

Il existe sept opérateurs de transformation qui requièrent un argument numérique *(numeric)* ou de type couple *(pair)*.

$$
\begin{aligned}
(x,\ y)\ \texttt{shifted}\ (a,\ b) &= (x+a,\ y+b);\\
(x,\ y)\ \texttt{rotated}\ \theta &= (x\cos\theta - y\sin\theta,\ x\sin\theta + y\cos\theta);\\
(x,\ y)\ \texttt{slanted}\ a &= (x+ay,\ y);\\
(x,\ y)\ \texttt{scaled}\ a &= (ax,\ ay);\\
(x,\ y)\ \texttt{xscaled}\ a &= (ax,\ y);\\
(x,\ y)\ \texttt{yscaled}\ a &= (x,\ ay);\\
(x,\ y)\ \texttt{zscaled}\ (a,\ b) &= (ax-by,\ bx+ay).
\end{aligned}
$$

6 Les opérateurs `arclength` et `arctime of` ne se trouvent que dans les versions 0.50 et supérieures de METAPOST

Ces opérations se comprennent toutes d'elles-mêmes sauf `zscaled` qui peut être comprise comme une multiplication par des nombres complexes. L'effet de `zscaled` $(a,\ b)$ est de produire une rotation et une homothétie de façon à transformer $(1,\ 0)$ en $(a,\ b)$. Le résultat de `rotated` θ est une rotation de θ degrés dans le sens trigonométrique.

Toute combinaison de translation, rotation, etc. est une transformation affine dont le résultat est de transformer tout couple $(x,\ y)$ en

$$(t_x + t_{xx}x + t_{xy}y,\ t_y + t_{yx}x + t_{yy}y),$$

pour un quelconque sextuplet $(t_x,\ t_y,\ t_{xx},\ t_{xy},\ t_{yx},\ t_{yy})$. Cette information peut être stockée dans une variable de type transformation de telle sorte que `transformed T` peut être équivalent à

```
xscaled -1 rotated 90 shifted (1,1)
```

si `T` est une variable de type transformation appropriée. La transformation `T` pourrait être ensuite initialisée par une expression de type transformation comme celle-ci :

```
transform T;
T = identity xscaled -1 rotated 90 shifted (1,1);
```

Comme l'indique l'exemple, les expressions de type transformation peuvent être construites en appliquant des opérations de transformation à d'autres transformations. La transformation prédéfinie `identity` est très utile comme point de départ pour un tel processus. Ceci peut être illustré en paraphrasant l'équation de `T` précédente en français : « `T` doit être la transformation obtenue en faisant ce que fait `identity`, puis mettre à l'échelle -1 la coordonnée x, effectuer une rotation de 90° et faire une translation de vecteur (1, 1) ». Ceci fonctionne parce que `identity` est la transformation identité qui ne fait rien : `transformed identity` est un « non-opérateur » ou « opérateur nul ».

La syntaxe pour les expressions de type transformation et pour les opérateurs de transformation est donnée dans la figure 27. Elle inclut deux options supplémentaires pour <transformation> :

`reflectedabout`$(p,\ q)$

qui crée la figure symétrique par rapport à la ligne définie par les deux points p et q ;

`rotatedaround`$(p,\ \theta)$

qui effectue une rotation de θ degrés dans le sens trigonométrique autour du point p.

Par exemple, l'équation d'initialisation de la transformation `T` aurait pu être

```
T = identity reflectedabout((2,0), (0,2))
```

<paire secondaire>→ <paire secondaire><transformation>
<chemin secondaire>→ <chemin secondaire><transformation>
<image secondaire>→ <image secondaire><transformation>
<stylo secondaire>→ <stylo secondaire><transformation>
<transformation secondaire>→ <transformation secondaire><transformation>

<transformation>→ `rotated`<numérique primaire>
`scaled`<numérique primaire>
`shifted`<paire primaire>
`slanted`<numérique primaire>
`transformed`<transformation primaire>
`xscaled`<numérique primaire>
`yscaled`<numérique primaire>
`zscaled`<paire primaire>
`reflectedabout`(<expression paire>,<expression paire>)
`rotatedaround`(<expression paire>,<expression numérique>)

FIG. 27 – Syntaxe des transformations et des opérateurs associés

Il existe aussi un opérateur unaire `inverse` qui requiert une transformation et en trouve une autre qui annule les effets de la première. Ainsi, si

$$p = q \texttt{ transformed } T$$

alors

$$q = p \texttt{ transformed inverse } T.$$

L'application de `inverse` à une transformation inconnue est illégale mais on déjà vu que l'on peut écrire

`T` = <expression transformation>

lorsque `T` n'a seulement été que déclarée sans avoir encore reçu de valeur. Il est aussi possible d'appliquer une transformation inconnue à une paire ou une transformation connue et d'utiliser le résultat dans une équation linéaire. Trois équations sont suffisantes pour déterminer une transformation.

Ainsi les équations

```
(0,1) transformed T' = (3,4);
(1,1) transformed T' = (7,1);
(1,0) transformed T' = (4,-3);
```

suffisent à METAPOST pour déterminer que la transformation `T'` est la combinaison d'une rotation et d'une homothétie

$$\begin{array}{rclrcl} t_{xx} & = & 4, & t_{xy} & = & -3, \\ t_{yx} & = & 3, & t_{yy} & = & 4, \\ t_x & = & 0, & t_y & = & 0. \end{array}$$

Les équations mettant en jeu une transformation inconnue sont traitées comme des équations linéaires des six paramètres qui définissent la transformation. Ces six paramètres peuvent être directement accessibles par

`xpart T`, `ypart T`, `xxpart T`, `xypart T`, `yxpart`, `yypart T`,

où `T` est une transformation. Par exemple, la figure 28 utilise les équations

```
xxpart T = yypart T; yxpart T = -xypart T
```

pour spécifier que la forme de `T` est préservée ; c'est-à-dire que c'est une combinaison d'une rotation, d'une translation et d'une homothétie (uniforme, c'est-à-dire sur les deux coordonnées simultanément).

8.4. Lignes discontinues

Le langage METAPOST offre de nombreux moyens pour changer l'apparence des lignes en plus du changement de leur épaisseur. Une façon de faire est d'utiliser des lignes pointillées comme le montrent les figures 5 et 23. La syntaxe pour ceci est

`draw`<expression chemin>`dashed` <motif de points>

où un <motif de points> est, en fait, un type spécial d'<expression dessin>. Il existe un <motif de points> prédéfini appelé `evenly` qui produit des pointillés longs de 3 points POSTSCRIPT séparés par des espaces de la même longueur. Un autre <motif de points> `withdots` produit des lignes de points séparés par 5 points POSTSCRIPT[7]. Pour obtenir des écarts plus importants entre les points ou les traits, l'échelle du <motif de points> peut être changée, comme dans la figure 29.

7 `withdots` ne se trouve que dans les version 0.5 et supérieures de METAPOST.

```
beginfig(28);
path p[];
p1 = fullcircle scaled .6in;
z1=(.75in,0)=-z3;
z2=directionpoint left of
p1=-z4;
p2 = z1..z2..{curl 1}z3..z4..{curl 1}cycle;
fill p2 withco-
lor .4[white,black];
unfill p1;
draw p1;
transform T;
z1 transformed T = z2;
z3 transformed T = z4;
xxpart T=yypart T;
yxpart T=-xypart T;
picture pic;
pic = currentpicture;
for i=1 upto 2 :
pic :=pic transformed T;
draw pic;
endfor
dotlabels.top(1,2,3);
dotlabels.bot(4);
endfig;
```

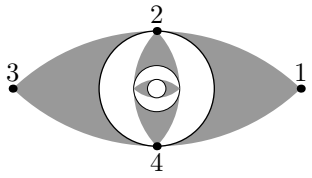


FIG. 28 – Code METAPOST et la « fractale » résultante

```
. . . . . dashed withdots scaled 2
..... dashed withdots
—— —— —— —— —— —— dashed evenly scaled 4
– – – – – – – – – – – – dashed evenly scaled 2
------------------------ dashed evenly
```

FIG. 29 – Lignes discontinues, (à droite, le <motif de points> ayant servi à sa construction)

Une autre manière de changer un <motif de points> est de modifier sa phase par décalage horizontal, phénomène illustré par la figure 30.

Lorsqu'on décale un motif de points, de telle sorte que l'axe y coupe le milieu d'un trait, le premier sera tronqué. Ainsi la ligne avec un motif `e4` commence avec un trait de longueur de 12 bp, est suivi par un blanc de 12 bp et un autre trait de 12 bp, etc., alors que

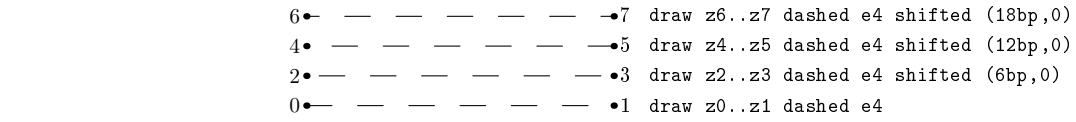


FIG. 30 – Lignes discontinues et instruction METAPOST permettant de les tracer (`e4` se réfère au motif de points `evenly scaled 4`)

`e4 shifted (-6 bp,0)` produit un trait de 6 bp, un blanc de 12 bp puis un trait de 12 bp, etc. Ce motif peut être spécifié plus directement par la fonction `dashpattern`

```
dashpattern(on 6bp off 12bp on 6bp)
```

Cela signifie « tracer les premiers 6 bp de la ligne, puis sauter les 12 bp suivants, puis tracer une nouvelle ligne de 6 bp et répéter ». Si la longueur de la ligne à mettre en pointillés est supérieure à 30 bp, les derniers 6 bp de la première copie du motif seront imbriqués avec les 6 bp du prochain morceau copié de façon à former un trait de 12 bp de long. La syntaxe générale pour la fonction `dashpattern` est indiquée dans la figure 31.

<motif de points>→ `dashpattern(`<liste on/off>`)`
<liste on/off>→ <liste on/off><clause on/off> | <clause on/off>
<clause on/off>→ `on`<tertiaire numérique>| `off`<tertiaire numérique>

FIG. 31 – Syntaxe de la fonction `dashpattern`

Puisque un motif de point *(dashpattern)* est en fait une sorte de figure spéciale, la fonction `dashpattern` renvoie une figure *(picture)*. Il n'est pas réellement nécessaire de connaître la structure d'une telle figure, et le lecteur occasionnel sautera certainement cette section (8.4). Pour ceux qui souhaitent en savoir plus, une petite expérience montre que si `d` est

```
dashpattern(on 6bp off 12bp on 6bp)
```

alors `llcorneer d` vaut (0,24) et `urcorner d` (24,24).

Dessiner `d` directement sans utiliser un motif de points produit deux segments horizontaux en trait fin. Comme ceci

Dans ce cas les lignes sont considérées comme ayant une épaisseur nulle (égale à zéro), mais ceci n'est pas très important puisque l'épaisseur est ignorée lorsqu'une figure est utilisée comme motif de trait.

La règle générale pour interpréter une figure d comme un motif de point (*dash pattern*) est que les segments dans d sont projetés sur l'axe des abscisses. Le motif résultant est répliqué à l'infini dans les deux directions en plaçant bout à bout les copies du motif. Les longueurs réelles des traits sont obtenues en partant depuis $x = 0$ et en cherchant dans la direction des x positifs.

Pour se faire une idée plus précise de la « réplication à l'infini », considérons $P(\mathsf{d})$ comme la projection de d sur l'axe des x et soit le vecteur $(P(\mathsf{d}), x)$, le résultat du décalage de d suivant x. Le motif résultant de la réplication à l'infini sera

$$\bigcup_{n\in\mathcal{N}} \text{shift}(P(\mathsf{d}), n \cdot l(\mathsf{d}))$$

où $l(\mathsf{d})$ mesure la longueur de $P(\mathsf{d})$. La définition la plus restrictive de cette longueur et $d_{max} - d_{min}$, où $[d_{max}, d_{min}]$ est l'intervalle de définition de la coordonnée x de $P(\mathsf{d})$. En fait, METAPOST utilise

$$\max(|y_0(\mathsf{d})|, d_{max} - d_{min}),$$

où $y_0(\mathsf{d})$ est la coordonnée y du contenu de d. Le contenu de d devrait s'allonger sur une ligne horizontale, mais si cela n'intervient pas, l'interpréteur de METAPOST prend juste la coordonnée y qui apparaît dans d.

Une figure utilisée comme un motif de points ne doit contenir ni texte ni région colorée, mais elle peut contenir des lignes qui sont elles-mêmes discontinues. Cela peut donner des petits pointillés à l'intérieur de plus grands, comme le montre la figure 32.

```
beginfig(32);
draw dashpattern(on 15bp off 15bp)
dashed evenly;
picture p;
p=currentpicture;
currentpicture :=nullpicture;
draw fullcircle scaled 1cm xsca-
led 3 dashed p;
endfig;
```

FIG. 32 – Code METAPOST et la sortie correspondante

8.5. Autres options

On aura pu noter que les lignes pointillées produites par `dashed evenly` apparaissent plus noires que blanches. Ceci est un effet du paramètre `linecap` qui contrôle

l'apparence de l'extrémité des lignes aussi bien que de celle des traits. Il existe aussi d'autres manières de modifier l'apparence de choses dessinées avec METAPOST.

Le paramètre `linecap` possède trois valeurs différentes, tout comme en POSTSCRIPT. Plain METAPOST donne à cette variable interne la valeur par défaut `rounded` qui produit un arrondissement des extrémités des segments comme le segment de `z0` à `z3`, dans la figure 33. Choisir `linecap :=butt` coupe les extrémités à ras de sorte que les traits produits par `dashed evenly` ont une longueur de 3 bp et non 3 bp plus l'épaisseur de la ligne. On peut également obtenir de extrémités carrées par le choix `linecap :=squared`, comme ce qui a été fait pour la ligne `z2` à `z5` dans la figure 33.

```
beginfig(33);
for i=0 upto 2 :
        z[i]=(0,-40i);
z[i+3]-z[i]=(100,30);
endfor
pickup pencircle scaled 18;
draw z0..z3 withcolor
.8white;
linecap :=butt;
draw z1..z4 withcolor
.8white;
linecap :=squared;
draw z2..z5 withcolor
.8white;
dotlabels.top(0,1,2,3,4,5);
endfig; linecap :=rounded;
```

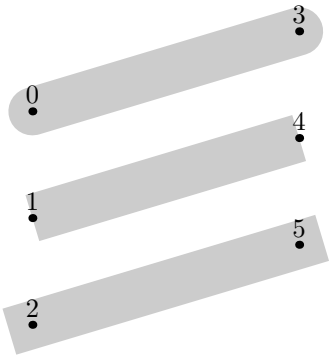


FIG. 33 – Code METAPOST et la sortie résultante

Un autre paramètre, emprunté au POSTSCRIPT, affecte la manière dont une déclaration `draw` traite les angles dans le tracé d'un chemin. Le paramètre `linejoin` peut être `rounded`, `beveled`, ou `mitered` comme le montre la figure 34. La valeur par défaut dans Plain METAPOST est `rounded` dont l'effet est celui du tracé par un pinceau circulaire.

Quand `linejoin` est mis à `mitered`, les angles engendrent une pointe, comme le montre la figure 35. Ceci pouvant être indésirable, il existe une variable interne appelée `miterlimit` qui contrôle comment les situations extrêmes de l'utilisation d'un `mitered` seront remplacées par un `beveled`. Pour Plain METAPOST, `miterlimit` a comme valeur par défaut `10.0` et les angles sont remplacés par l'aspect obtenu avec `beveled` lorsque le rapport de la longueur de raccordement sur la largeur de la ligne dépasse cette valeur.

```
beginfig(34);
for i=0 upto 2 :
   z[i]=(0,-50i); z[i+3]-
z[i]=(60,40);
   z[i+6]-z[i]=(120,0);
endfor
pickup pencircle scaled 24;
draw z0 -- z3 -- z6 withco-
lor .8white;
linejoin :=mitered;
draw z1..z4 -- z7 withcolor
.8white;
linejoin :=beveled;
draw z2..z5 -- z8 withcolor
.8white;
dotlabels.bot(0,1,2,3,4,5,6,7,8);
endfig; linejoin :=rounded;
```

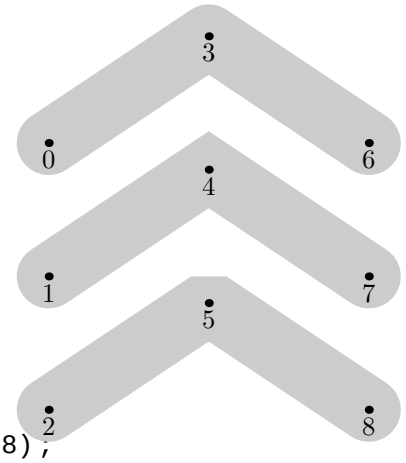


FIG. 34 – Code METAPOST et la sortie résultante

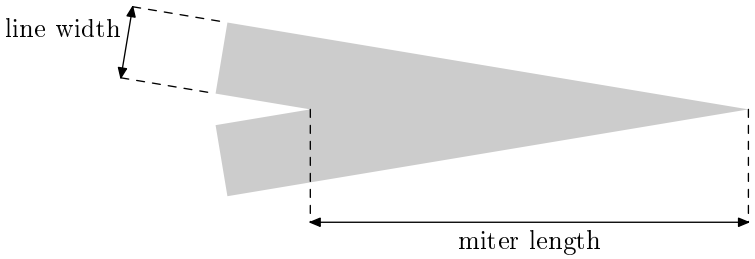


FIG. 35 – La longueur de raccordement et l'épaisseur de ligne dont le rapport est limité par `miterlimit`

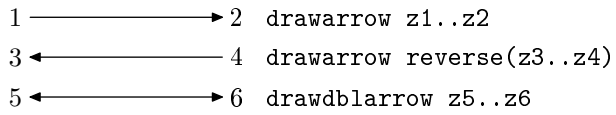


FIG. 36 – Trois moyens de tracer des flèches

Les paramètres `linecap`, `linejoin` et `miterlimit` sont particulièrement importants car ils affectent des éléments qui sont dessinés en arrière plan. Par exemple, Plain METAPOST possède des déclarations pour le tracé de flèches, dont les têtes de flèches sont légèrement arrondies lorsque `linejoin` a la valeur `rounded`. L'effet dépend de

l'épaisseur de la ligne et devient presque subtil pour une épaisseur de ligne par défaut de 0.5 bp, comme le montre la figure 36.

Le dessin de flèches, comme une de celles présentées dans la figure 36, s'obtient simplement par la commande

`drawarrow`<expression chemin>

au lieu de `draw`<expression chemin>. Ceci dessine le chemin indiqué en ajoutant un flèche au dernier point du chemin. Si l'on souhaite mettre la flèche au début du chemin, il suffit d'utiliser l'opérateur unaire `reverse` pour remplacer le chemin original par un nouveau dont le paramètre t sera inversé ; donc pour un chemin `p` dont `length p`$= n$,

`point` t `of reverse p` et `point` $n - t$ `of p`

sont équivalents.

Comme l'indique la figure 36, une déclaration commençant par

`drawdblarrow`<expression chemin>

dessine une double flèche. La taille de la tête de la flèche est garantie plus grande que l'épaisseur de ligne, mais il se peut qu'il faille l'ajuster si la ligne est très épaisse. Ceci est obtenu en assignant une nouvelle valeur à la variable interne `ahlength` qui détermine la longueur de la tête de la flèche (cf. figure 37). L'augmentation de `ahlength` à partir de la valeur par défaut de 4 points POSTSCRIPT jusqu'à 1,5 cm produit la grande tête de flèche que montre la figure 37. Il existe également un paramètre `ahangle` qui contrôle l'angle au somme de la tête de flèche. La valeur par défaut de cet angle est de 45 degrés, comme le montre la figure 37.

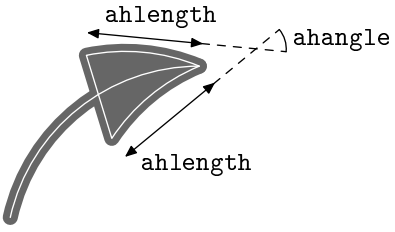


FIG. 37 – Une grande tête de flèche avec l'affichage des paramètres clés et, en blanc, les chemins utilisés pour la dessiner

La tête de la flèche est créée en remplissant la région triangulaire qui est soulignée par la ligne blanche dans la figure 37 et enfin entourée avec le stylo courant. Cette combinaison de remplissage et de tracé peut être condensée dans une déclaration unique `filldraw`

`filldraw`<expression chemin><clauses `dashed`, `withcolor` et `withpen` optionnelles>

L'<expression chemin> doit être un chemin fermé comme le chemin triangulaire de la figure 37. Ce chemin ne doit pas être confondu avec l'argument de type chemin de `drawarrow` qui, lui, est indiqué par la ligne blanche dans la figure.

Les lignes blanches, telles que celles de la figure 37, peuvent être créées par une déclaration `undraw`. Cette déclaration est une version d'effacement de `draw` qui dessine avec l'option `withcolor background` comme le ferait la déclaration `unfill`. Il existe également une déclaration `unfilldraw`, juste au cas où quelqu'un en aurait besoin.

Les déclaration `filldraw`, `undraw` et `unfilldraw` et toutes les déclarations de dessin de flèches sont similaires aux déclarations `fill` et `draw` qui peuvent prendre les options `dashed`, `withcolor` et `withpen`. Lorsqu'on a beaucoup de déclarations de dessins, il peut-être agréable d'appliquer une option telle que `withcolor 0.8white` à tous les dessins sans avoir à taper ceci de façon répétitive comme ce qui a été fait dans les figures 33 et 34. La déclaration permettant de le faire est

`drawoptions(`<texte>`)`

où l'argument <texte> contient une séquence de `dashed`, `withcolor` et `withpen` qui sera appliquée automatiquement à toutes les déclarations de dessin. Si on spécifie

```
drawoptions(withcolor .5[black,white])
```

et que l'on souhaite ensuite tracer une ligne noire, on peut surcharger la macro `drawoptions` en indiquant

`draw`<expression chemin>`withcolor black`

Pour désactiver toutes les `drawoptions` ensembles, il suffit de donner une liste vide

```
drawoptions()
```

Ce qui est fait automatiquement par la macro `beginfig`.

Puisque les options hors de propos sont ignorées, on peut toujours écrire une déclaration du genre

```
drawoptions(dashed evenly)
```

suivie d'une séquence de commandes `draw` et `fill`.

Si on utilise un motif de points pour un remplissage qui n'a pas de sens, le paramètre `dashed evenly` sera ignoré pour la commande `fill`. Il se trouve que

`drawoption(withpen` <expression stylo>`)`

n'affecte pas les déclarations `fill` et `draw`. En fait, il existe une variable stylo spéciale appelée `currentpen` telle que `fill... withpen currentpen` est équivalent à la déclaration `filldraw`.

Plus précisément que signifie de dire que les options de dessins affectent les déclarations où elles sont pertinentes ?

L'option `dashed` <motif de point> affecte seulement les déclarations

`draw` <expression chemin>

et le texte apparaissant dans l'argument <expression image> des déclarations

`draw` <expression image>

est seulement affecté par l'option `withcolor` <expression couleur>. Pour toutes les autres combinaisons de déclaration de dessins et d'options, il y a quelques effets. Une option appliquée à une déclaration `draw` <expression image> affectera en général des parties de la figure mais non les autres. Par exemple, une option `dashed` ou `withpen` affectera toutes les lignes dans la figure mais pas les étiquettes.

8.6. Stylos ou plumes *(pens)*

Les sections précédentes donnent de nombreux exemples de `pickup` <expression stylo> et `withpen` <expression stylo>, mais il n'y avait aucun exemple de plumes autre que

`pencircle scaled`<numérique primaire>

qui produit des lignes d'une épaisseur donnée. Pour obtenir des effets calligraphiques tels que dans la figure 38, on peut appliquer n'importe quel opérateur de transformation exposé dans la section (8.3). Le point de départ pour de telles transformations est `pencircle`, un cercle de diamètre d'un point POSTSCRIPT. Ainsi des transformations affines produisent une forme de plume circulaire ou elliptique. L'épaisseur des lignes tracées avec la plume dépend de la perpendicularité de la ligne avec le grand axe de l'ellipse.

La figure 38 montre que les opérateurs `lft`, `rt`, `top` et `bot` répondent à la question : « si la plume courante est située à la position indiquée par l'argument, où sera son angle gauche, droit, haut ou bas ? » Dans ce cas, la plume courante est l'ellipse indiquée dans la déclaration `pickup` et sa *bounding box* est large de 0.1734 pouces et haute de 0.1010 pouces, de telle sorte que `rt x3` égale `x3+0.0867in` et `bot y5` égale `y5-0.0505in`. Les opérateurs `lft`, `rt`, `top` et `bot` acceptent aussi des arguments de type couple pour lesquels ils calculent les coordonnées x et y du point de la forme de la plume le plus à gauche, le plus à droite, le plus haut ou le plus bas. Par exemple

$$\texttt{rt}(x,\ y) = (x,\ y) + (0.0867\text{ in},\ 0.0496\text{ in})$$

pour la plume dans la figure 38. La déclaration `beginfig` remet la plume courante à sa valeur par défaut qui est

```
pencircle scaled 0.5bp
```

au début de chaque figure. Cette valeur peut être re-sélectionnée à n'importe quel moment par la commande `pickup defaultpen`.

Ceci serait la fin de l'histoire consacrée aux plumes, sauf que pour des raisons de compatibilité avec METAFONT, METAPOST permet également aux plumes d'être polygonales. Il existe une plume prédéfinie appelée `pensquare` qui peut être transformée pour produire des plumes en forme de parallélogramme. En fait, il existe même un opérateur appelé `makepen` qui fabrique la taille et la forme à partir d'un chemin de forme polygonale convexe. Si le chemin n'est pas exactement convexe ou polygonal, l'opérateur `makepen` redressera les côtés et/ou déplacera quelques sommets. En particulier, `pensquare` est équivalent à

```
makepen((-.5,-.5) -- (.5,-.5) -- (.5,.5) -- (-.5,.5) -- cycle)
```

L'inverse de `makepen` est l'opérateur `makepath` qui requiert un <stylo primaire> et renvoie le chemin correspondant. Ainsi `makepath pencircle` produit un chemin circulaire identique à `fullcircle`. Ceci fonctionne aussi pour une plume polygonale de telle sorte que

`makepath makepen`<expression chemin>

prendra n'importe quel chemin cyclique et le transformera en un polygone convexe.

```
beginfig(38);
pickup pencircle scaled .2in yscaled
.08 rotated 30;
x0=x3=x4;
z1-z0 = .45in*dir 30;
z2-z3 = whatever*(z1-z0);
z6-z5 = whatever*(z1-z0);
z1-z6 = 1.2*(z3-z0);
rt x3 = lft x2;
x5 = .55[x4,x6];
y4 = y6;
lft x3 = bot y5 = 0;
top y2 = .9in;
draw z0 -- z1 -- z2 -- z3 -- z4 --
z5 -- z6 withcolor .7white;
dotlabels.top(0,1,2,3,4,5,6);
endfig;
```

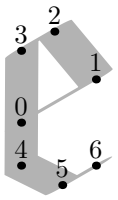


FIG. 38 – Code METAPOST et la figure « calligraphique » résultante

8.7. Découpage et commandes bas niveau de tracé

Les instructions de tracé telles que `draw`, `fill` et `unfill` font partie du package des macros de plain METAPOST et sont définies en termes de « primitives ». La principale différence entre les déclarations de dessins présentées dans les sections précédentes et les versions « primitives » est que les déclarations de dessins nécessitent toutes de spécifier une variable de type figure pour stocker le résultat. Pour `draw`, `fill` et les déclarations associées, le résultat est toujours stocké dans une variable de type figure appelée `currentpicture`. La syntaxe pour les déclarations des primitives de tracé qui permettent de spécifier une variable de type figure est indiquée dans la figure 39.

<commande addto>→
    `addto`<variable image>`also`<expression image><liste d'options>
    |`addto`<variable image>`contour`<expression chemin><liste d'options>
    |`addto`<variable image>`doublepath` <expression chemin><liste d'options>
<liste d'options>→ <vide> |<options de tracé><liste d'options>
<options de tracé>→ `withcolor`<expression couleur>
    |`withpen`<expression stylo> |`dashed`<expression image>

FIG. 39 – Syntaxe des déclarations des primitives de tracé

La syntaxe des commandes primitives de tracé est compatible avec METAFONT . Le tableau 2 montre comment les déclarations des primitives de tracé sont reliées aux ins-

tructions familières `draw` et `fill`. Chacune des déclarations dans la première colonne du tableau peut être terminée par une <liste d'options>, ce qui équivaut à ajouter cette <liste d'options> aux entrées correspondantes de la seconde colonne du tableau. Par exemple,

`draw` p `withpen pencircle`

est équivalent à

`addto currentpicture doublepath` p `withpen currentpen withpen pencircle`

où `currentpen` est une plume spéciale qui stocke toujours la dernière plume retenue. La seconde option `withpen` surcharge silencieusement le `withpen currentpen` à partir du développement de `draw`.

TAB. 2 – Déclarations communes de dessin et versions de primitive équivalente : q est la plume courante *(currentpen)*, b pour `background`, p est un chemin quelconque, c est un chemin fermé et pic est une <expression dessin>

Instruction	Primitive équivalente
`draw` pic	`addto currentpicture also` pic
`draw` p	`addto currentpicture doublepath` p `withpen` q
`fill` c	`addto currentpicture contour` c
`filldraw` c	`addto currentpicture contour` c `withpen` q
`undraw` pic	`addto currentpicture also` pic `withcolor` b
`undraw` p	`addto currentpicture doublepath` p `withpen` q `withcolor` b
`unfill` c	`addto currentpicture contour` c `withcolor` b
`unfilldraw` c	`addto currentpicture contour` c `withpen` q `withcolor` b

Il existe deux primitives de dessin supplémentaires qui n'acceptent aucune option de dessin. La première est la commande `setbounds` qui a été vue dans la section 7.3 ; la seconde est la commande `clip` :

`clip`<variable image>`to`<expression chemin>

Soit un chemin cyclique ; l'instruction précédente découpe le contenu de la variable figure pour éliminer tout ce qui est en dehors du chemin fermé.

Il n'existe pas de version « haut niveau » de cette déclaration, de telle sorte qu'il faut utiliser :

`clip currentpicture to`<expression chemin>

si l'on veut découper la `currentpicture`. La figure 40 illustre ce découpage.

```
beginfig(40);
path p[];
p1 = (0,0){curl 0}..(5pt,-3pt)..{curl
0}(10pt,0);
p2 = p1..(p1 yscaled-1 shifted(10pt,0));
p0 = p2;
for i=1 upto 3 :
    p0 :=p0.. p2 shifted (i*20pt,0);
endfor
for j=0 upto 8 :
    draw p0 shifted (0,j*10pt);
endfor
p3 = fullcircle shifted (.5,.5) scaled
72pt;
clip currentpicture to p3;
draw p3;
endfig;
```

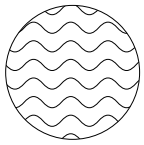


FIG. 40 – Code METAPOST et la figure découpée qui en résulte

Toutes les opérations primitives de dessin serait inutiles sans la dernière opération appelée `shipout`. La déclaration

`shipout`<expression image>

transforme la figure en un fichier POSTSCRIPT dont le nom se termine par *.nnn*, où `nnn` est la représentation décimale de la valeur de la variable interne `charcode` ne nom « `charcode` » est choisi pour la compatibilité avec METAFONT. Normalement, `beginfig` définit `charcode` et `endfig` invoque `shipout`.

9. Les macros

Comme il a été évoqué précédemment, METAPOST possède un ensemble de macros automatiquement chargées, qui s'appelle le package des macros Plain. Certaines des commandes présentées dans les sections précédentes sont définies comme des macros au lieu d'être des instructions du langage METAPOST. Le propos de ce chapitre est d'expliquer comment écrire de telles macros.

Les macros sans arguments sont très simples. Une définition de macro

`def`<*token* symbolique> = <texte de remplacement> `enddef`

permet d'indiquer qu'un <*token* symbolique> est une abréviation pour un <texte de remplacement>, où le <texte de remplacement> peut être virtuellement une séquence quelconque de *tokens*. Par exemple, le package Plain définit la déclaration `fill` presque comme ceci

```
def fill = addto currentpicture contour enddef
```

Les macros avec des arguments sont semblables sauf qu'elles ont des paramètres formels qui disent comment utiliser les arguments dans le <texte de remplacement>. Par exemple, la macro `rotatedaround` est définie comme ceci

```
def rotatedaround(expr z, d) = shifted -z rotated d shifted z
                               enddef;
```

Le terme `expr` dans cette définition signifie que les paramètres formels `z` et `d` peuvent être des expressions arbitraires. Ils doivent être des expressions de type paire mais l'interpréteur de METAPOST ne le vérifie pas immédiatement.

Puisque METAPOST est un langage interprété, les macros avec arguments ressemblent beaucoup à des sous-programmes. Les macros METAPOST sont souvent utilisées comme des sous-programmes, de telle sorte que le langage inclut des concepts de programmation pour supporter cela. Ces concepts incluent les variables locales, les boucles et les choix conditionnels.

9.1. Groupes

Le groupe, dans METAPOST, est essentiel pour les fonctions et les variables locales. L'idée fondamentale est qu'un groupe est une séquence de déclarations pouvant être suivie par une expression permettant à certains *tokens* symboliques de retrouver leur signification initiale à la fin du groupe. Si un groupe se termine avec une expression, le groupe se comporte comme une fonction qui retourne cette expression. Autrement, le groupe n'est juste qu'une déclaration composée.

La syntaxe pour un groupe est

`begingroup`<liste d'instructions> `endgroup`

ou

`begingroup`<liste d'instructions><expression>`endgroup`

où la <liste d'instructions> est une séquence de déclarations, suivie chacune par un point-virgule. Un groupe avec une <expression> après la <liste d'instructions> se comporte comme un <primaire> dans la figure 14 ou comme un <atome numérique> dans la figure 15.

Puisque le <texte de remplacement> pour la macro `beginfig` commence avec `begingroup` et que le <texte de remplacement> pour `endfig` se termine avec `endgroup`, chaque figure dans un fichier METAPOST se comporte comme un groupe. Ceci permet aux figures d'avoir des variables locales. Dans la section 6.2, on a vu que les noms de variables commençant par `x` ou `y` correspondant à des variables locales dans le sens qu'elles ne sont pas initialisées au début de chaque figure et que leur valeur est perdue à la fin de chaque figure. L'exemple suivant illustre le fonctionnement des variables locales.

```
x23 = 3.1;
beginfig(17);
     ⋮
y3a=1; x23=2;
     ⋮
endfig;
show x23, y3a;
```

Le résultat de la commande `show` est

```
» 3.1
» y3a
```

indiquant que `x32` a retrouvé sa valeur initiale de `3.1` et que `y3a` est complètement inconnue comme elle l'était lors de la déclaration `beginfig(17)`.

Le caractère local des variables `x` et `y` est obtenu par la déclaration

`save x, y`

dans le <texte de remplacement> de `beginfig`.

En général, les variables sont rendues locales par la déclaration

`save`<liste de *tokens* symboliques>

où la <liste de *tokens* symboliques> est une liste de *tokens* séparés par des virgules

<liste de *tokens* symboliques>→ <*token* symbolique>
| <*token* symbolique>, <liste de *tokens* symboliques>

Toutes les variables dont le nom commence par l'un des *tokens* symboliques spécifiques deviennent des inconnues numériques et leur valeur actuelle est sauvée pour être restaurée à la fin du groupe courant. Si la déclaration `save` est utilisée en dehors d'un groupe, la valeur originale est simplement effacée.

Le propos principal de la déclaration `save` est de permettre aux macros d'utiliser des variables sans interférer avec des variables existantes ou des variables lors d'autres appels de la même macro. Par exemple, la macro prédéfinie `whatever` possède le <texte de remplacement>

```
begingroup save ?; ? endgroup
```

Cette instruction retourne une quantité numérique inconnue, mais elle n'est pas appelée longtemps « ? » puisque ce nom est local au groupe. Demander le nom par `show whatever` donne

```
» %CAPSULEnnnn
```

où *nnnn* est un nombre d'identification qui est choisi quand `save` fait disparaître le point d'interrogation.

En dépit de sa versatilité, `save` ne peut pas être utilisée pour effectuer des changements locaux de variables internes de METAPOST. Une déclaration telle que

```
save linecap
```

obligerait METAPOST à oublier temporairement la signification particulière de cette variable et à la considérer juste comme une inconnue numérique. Si l'on veut tracer une ligne pointillée avec `linecap :=butt` et ensuite revenir à la valeur précédente, on peut utiliser la déclaration `interim` comme suit

```
begingroup interim linecap :=butt;
draw<expression chemin>dashed evenly; endgroup
```

Ceci sauvegarde la valeur de la variable interne `linecap` et permet de lui donner temporairement une nouvelle valeur sans perdre le fait que `linecap` est une variable interne. La syntaxe générale est

`interim`<variable interne> `:=` <expression numérique>

9.2. Macros paramétrées

L'idée de base derrière les macros paramétrées est d'obtenir une plus grande souplesse en autorisant de passer une information auxiliaire à une macro. On a déjà vu que les définitions de macros peuvent avoir des paramètres formels qui représentent des expressions transmises lorsque la macro est appelée. Pour le moment, une définition telle que

`def rotatedaround(expr z, d) =` <texte de remplacement> `enddef`

permet à l'interpréteur METAPOST de comprendre des appels de macro sous la forme

`rotatedaround(`<expression>`,`<expression>`)`

Le mot clef `expr` dans la définition de la macro signifie que les paramètres peuvent être des expressions de n'importe quel type.

Quand la définition spécifie `(expr z, d)`, les paramètres formels `z` et `d` se comportent comme des variables de type approprié. À l'intérieur du <texte de remplacement>, ils peuvent être utilisés dans des expressions exactement comme des variables, mais ces dernières ne peuvent pas être redéclarées ou assignées. Il n'y a aucune restriction contre des arguments inconnus, en partie ou en totalité. Ainsi, la définition

```
def midpoint(expr a, b) = (.5[a,b]) enddef
```

fonctionne parfaitement quand `a` et `b` sont inconnues. Une équation telle que

```
midpoint(z1,z2)=(1,1)
```

peut être utilisée pour aider à la détermination de `z1` et `z2`.

Il faut noter que la définition précédente de `midpoint` fonctionne pour des variables numériques, des couples ou des couleurs aussi longtemps que les deux paramètres sont du même type. Si pour une raison quelconque, on veut qu'une macro `middlepoint` fonctionne avec un chemin ou une figure *(picture)*, il sera nécessaire d'effectuer un test `if` sur le type des argument. Ceci utilise le fait qu'il existe l'opérateur unaire

`path`<primaire>

qui retourne un résultat booléen indiquant si l'argument est un chemin.

Puisque le test élémentaire `if` a la syntaxe

`if`<expression booléenne> :<*tokens* équilibrés> `else :`<tokens équilibrés>`fi`

où les <*tokens* équilibrés> peuvent être n'importe quelle séquence de *tokens* respectant les emboîtements de `if` et `fi`, la macro `middlepoint` complète avec un test sur les types ressemblera à

```
def middlepoint(expr a) = if path a : (point .5*length a of a)
        else : .5(llcorner a + ucorner a) fi enddef;
```

La syntaxe complète pour les tests `if` est indiquée dans la figure 41. Elle permet des tests `if` multiples comme

`if` e_1 : ... `else :` `if` e_2 : ... `else :` ... `fi fi`

qui peuvent être raccourcis par

`if` e_1 : ... `elseif` e_2 : ... `else :` ... `fi`

où e_1 et e_2 représentent des expressions booléennes.

Il faut noter que les tests `if` ne sont pas des instructions et les <*tokens* équilibrés> dans les règles syntaxiques peuvent être n'importe quelle séquence de *tokens* équilibrés même si elles ne forment pas une expression complète ni une instruction complète. Ainsi, on aurait pu économiser deux *tokens*, au dépend de la clarté, en définissant `midpoint` comme ceci

```
def midpoint(expr a) = if path a : (point .5*length a of
     else : .5(llcorner a + urcorner fi a) enddef;
```

<test if> → `if`<expression booléenne> :<*tokens* équilibrés><alternatives>`fi`
<alternatives>→ <vide>
|`else :`<*tokens* équilibrés>
|`elseif`<expression booléenne> :<*tokens* équilibrés><alternatives>

FIG. 41 – La syntaxe des tests `if`

Le but réel des macros et des tests `if` est d'automatiser les tâches répétitives et de permettre de résoudre séparément des tâches secondaires importantes. Par exemple, on utilise les macros `draw_marked`, `mark_angle` et `mark_rt_angle` pour marquer les lignes et les angles qui apparaissent dans la figure 42.

La tâche de la macro `draw_marked` est de tracer un chemin avec un nombre donné de marques à proximité de ses points milieux. Un point de départ commode du problème est la résolution du sous-problème du tracé d'une seule marque perpendiculaire au chemin `p` pour une valeur quelconque du « temps » `t`.

La macro `draw_mark` réalise ceci dans la figure 43 en déterminant d'abord un vecteur `dm` normal à `p` en `t`. Pour simplifier le positionnement du trait, la macro `draw_marked` est définie pour prendre une longueur d'arc `a` le long de `p` et d'utiliser l'opérateur `arctime` pour calculer `t`.

À l'aide de la résolution du sous-problème du tracé d'une seule marque en dehors du chemin, la macro `draw_marked` a uniquement besoin de tracer le chemin et d'appeler `draw_mark` avec les valeurs appropriées de la longueur de l'arc. La macro `draw_marked`, dans la figure 43, utilise `n` valeurs équidistantes de `a` centrées sur `.5*arclength p`.

```
beginfig(42);
pair a, b, c, d;
b=(0,0); c=(1.5in,0); a=(0,.6in);
d-c = (a-b) rotated 25;
dotlabel.lft("a",a);
dotlabel.lft("b",b);
dotlabel.bot("c",c);
dotlabel.llft("d",d);
z0=.5[a,d];
z1=.5[b,c];
(z.p-z0) dotprod (d-a) = 0;
(z.p-z1) dotprod (c-b) = 0;
draw a--d;
draw b--c; draw z0--z.p--z1;
draw_marked(a--b, 1);
draw_marked(c--d, 1);
draw_marked(a--z.p, 2);
draw_marked(d--z.p, 2);
draw_marked(b--z.p, 3);
draw_marked(c--z.p, 3);
mark_angle(z.p, b, a, 1);
mark_angle(z.p, c, d, 1);
mark_angle(z.p, c, b, 2);
mark_angle(c, b, z.p, 2);
mark_rt_angle(z.p, z0, a);
mark_angle(z.p, b, z1, b);
endfig;
```

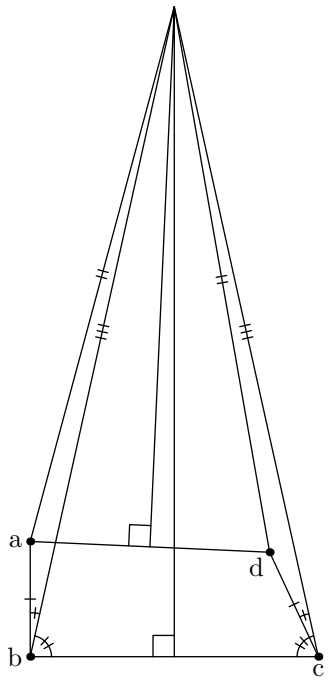


FIG. 42 – Code METAPOST et la figure correspondante

```
marksize=4pt;

def draw_mark(expr p, a) =
    begingroup
    save t, dm; pair dm;
    t = arctime a of p;
    dm = marksize*unitvector direction t of p
       rotated 90;
    draw (-.5dm.. .5dm) shifted point t of p;
    endgroup
enddef;

def draw_marked(expr p, n) =
    begingroup
    save amid;
    amid = .5*arclength p;
    for i=-(n-1)/2 upto (n-1)/2 :
       draw_mark(p, amid+.6marksize*i);
    endfor
    draw p; endgroup
    enddef;
```

FIG. 43 – Macro pour le tracé d'un chemin `p` avec `n` marques perpendiculaires

```
angle_radius=8pt;

def mark_angle(expr a, b, c, n) =
   begingroup
   save s, p; path p;
   p = unitvector(a-b){(a-b)rotated 90 }..
   unitvector(c-b);
   s = .9marksize/length(point 1 of p - point 0 of p);
   if s<angle_radius : s :=angle_radius; fi
   draw_marked(pscaled s shifted b, n);
   endgroup
enddef;

def mark_rt_angle(expr a, b, c) =
   draw ((1,0)--(1,1)--(0,1))
     zscaled (angle_radius*unitvector(a-b)) shifted b
enddef;
```

FIG. 44 – Macro pour le marquage des angles

Puisque la macro `draw_marked` fonctionne pour des courbes, elle peut être utilisée pour tracer des arcs que la macro `mark_angle` crée. Soient les points `a`, `b` et `c` qui définissent un angle dans le sens trigonométrique en `b`, la macro `mark_angle` nécessite de créer un petit arc `p` reliant le segment `ba` au segment `bc`. La définition de la macro de la figure 44 fait ceci en créant un arc `p` de rayon unitaire et ensuite en calculant un facteur d'échelle `s` qui le rend suffisamment grand pour être vu clairement.

La macro `mark_rt_angle` est beaucoup plus simple. Elle considère un coin d'angle droit et utilise l'opérateur `zscaled` pour le tourner et le mettre à l'échelle comme nécessaire.

9.3. Suffixes et paramètres textes

Les paramètres de macros ne sont pas nécessairement toujours des expressions comme celles des exemples précédents. Le remplacement du mot-clé `expr` par `suffix` ou `text` dans une définition de macro déclare les paramètres comme étant des noms de variables ou des séquences arbitraires de *tokens*. Par exemple, il existe une macro prédéfinie appelée `hide` qui prend un paramètre texte et l'interprète comme une séquence d'instructions produisant finalement un <texte de remplacement> vide. En d'autres termes, `hide` exécute ses arguments et enfin donne le *token* suivant comme si rien ne s'était passé.

Ainsi

```
show hide(numeric a,b; a+b=3; a-b=1) a;
```

imprime « >> `2.` »

Si la macro `hide` n'était pas prédéfinie, elle pourrait être définie comme ceci

```
def ignore(expr a) = enddef;
def hide(text t) = ignore(begingroup t; 0 endgroup) enddef;
```

La déclaration représentée par le paramètre texte `t` serait évaluée comme une partie du groupe qui forme l'argument de la macro `ignore`. Puisque la macro `ignore` possède un <texte de remplacement> vide, le développement de la macro `hide` ne produit finalement rien.

Un autre exemple de macro prédéfinie avec un paramètre texte est `dashpattern`. La définition de `dashpattern` commence par

```
def dahspattern (text t) =
   begingroup save on, off;
```

puis définit `on` et `off` comme des macros qui créent la figure désirée quand le paramètre texte `t` apparaît dans le <texte de remplacement>.

Les paramètres textes sont très généraux, mais leur généralité est parfois gênante. Si l'on veut juste passer un nom de variable dans une macro, il est préférable de le déclarer comme un paramètre suffixe. Par exemple

```
def incr(suffix $) = begingroup $ :=$+1; $ endgroup enddef;
```

définit une macro qui prend n'importe quelle variable numérique, lui ajoute 1 et retourne la nouvelle valeur. Puisque les noms de variables peuvent être plus long qu'un seul *token*

```
incr(a3b)
```

est parfaitement acceptable si `a3b` est une variable numérique. Les paramètres suffixes sont visiblement plus généraux que les noms de variables parce que la définition dans la figure 16 permet à un <suffixe> de démarrer avec un <indice>.

La figure 45 montre comment les paramètres `suffix` et `expr` peuvent être utilisés ensemble. La macro `getmid` prend une variable de type chemin et crée des tableaux de points et de directions dont les noms sont obtenus en ajoutant `mid`, `off` et `dir` à la variable de type chemin. La macro `joinup` prend des tableaux de points et de directions et crée un chemin de longueur `n` qui passe par chaque `pt[i]` avec les directions `d[i]` ou `-d[i]`.

Une définition commençant par

```
def joinup(suffix pt, d)(expr n) =
```

pourrait suggérer que des appels de la macro `joinup` doivent avoir deux jeux de parenthèses comme

```
joinup(p.mid, p.dir)(36)
```

au lieu de

```
joinup (p.mid, p.dir, 36)
```

En fait, les deux formes sont possibles. Les paramètres, dans un appel de macro, doivent être séparés par des virgules ou par des paires de parenthèses «)(». La seule restriction est qu'un paramètre texte doit être suivi par une parenthèse droite.

Par exemple, une macro `foo` avec un paramètre texte et un paramètre `expr` peut être appelée par

```
foo(a,b)(c)
```

auquel cas le paramètre texte est « `a,b` » et `c` est le paramètre `expr`, mais

```
foo(a,b,c)
```

considère « `a,b,c` » comme paramètre texte et laisse l'interpréteur METAPOST rechercher le paramètre `expr`.

9.4. Macros *vardef*

Une définition de macro peut commencer avec `vardef` au lieu de `def`. Les macros définies de cette manière sont appelées des macros `vardef`. Elles sont particulièrement bien adaptées pour des applications dans lesquelles les macros sont utilisées en guise de fonctions ou de sous-routines. L'idée principale est qu'une macro `vardef` est comme une variable du type « macro ».

Au lieu de `def` <*token* symbolique>, une macro `vardef` commence par

`vardef`<variable générique>

où une <variable générique> est un nom de variable avec un indice numérique remplacé par le symbole générique d'indice [ ]. En d'autres termes, le nom suivant `vardef` obéit exactement à la même syntaxe que celle des noms donnés dans une déclaration de variable. Il s'agit d'une séquence de *tags* et de symboles d'indice générique commençant avec un *tag*, où le *tag* est un *token* symbolique qui n'est ni une macro ni une primitive comme cela est expliqué dans la section 6.2.

Le cas le plus simple est celui d'un nom de macro `vardef` constitué d'un simple *tag*. Dans de telles circonstances, `def` et `vardef` procurent approximativement les mêmes fonctionnalités. La différence la plus évidente est que `begingroup` et `endgroup` sont automatiquement insérés au début et à la fin du <texte de remplacement> dans chaque macro `vardef`. Ceci considère le <texte de remplacement> comme un groupe de telle sorte qu'une macro vardef se comporte comme un appel de sous-routine ou de fonction.

Une autre propriété des macros vardef est qu'elles permettent des noms de macro multi-*tokens* et des noms mettant en jeu des indices génériques. Lorsque le nom d'une macro vardef possède des indices génériques, il faut donner des valeurs numériques lorsque la macro est appelée.

Après une définition de macro

```
vardef a[]b(expr p) = <texte de remplacement> enddef;
```

`a2b((1,2))` et `a3b((1,2)..(3,4))` sont des appels de macro.

Mais comment le <texte de remplacement> peut-il faire la différence entre `a2b` et `a3b`? Deux paramètres suffixes implicites sont fournis à ce propos. Chaque macro vardef possède les paramètres `#@` et `@`, où `@` est le dernier *token* dans le nom lors du dernier appel et `#@` correspond à tout ce qui précède le dernier *token*. Ainsi `#@` correspond à `a2` quand le nom donné est `a2b` et `a3` lorsque le nom est `a3b`.

Supposons, par exemple, que la macro `a[]b` doive prendre son argument et le translater d'une quantité qui dépend du nom de la macro. La macro pourrait être définie comme ceci

```
vardef a[]b(expr p) = p shifted(#@,b) enddef;
```

Ainsi `a2b((1,2))` signifierait `(1,2) shifted (a2,b)` et `a3b((1,2)..(3,4))` signifierait `((1,2)..(3,4)) shifted (a3,b)`.

Si la macro avait été `a.b[]`, `#@` serait toujours `a.b` et le paramètre `@` donnerait l'indice numérique. Ainsi `a@` se référerait à un élément du tableau `a[]`. Il faut noter que `@` est un paramètre suffixe et non un paramètre expression, de telle sorte qu'une expression comme `@+1` serait illégale. Le seul moyen pour attribuer des valeurs numériques d'un indice dans un paramètre suffixe est de les extraire de la chaîne de caractères renvoyée par l'opérateur `str`. Cet opérateur nécessite un suffixe et renvoie une représentation chaîne de caractère d'un suffixe. Donc `str @` serait `"3"` dans `a.b3` et `"3.14"` dans `a.b3.14` ou `a.b[3.14]`. Puisque la syntaxe pour un <suffixe> dans la figure 16 nécessite que les indices négatifs soient entre crochets, `str @` renvoie `"[-3]"` dans l'appel `a.b[-3]`.

L'opérateur `str` est généralement réservé à des utilisations de dernier recours. Il est préférable d'utiliser des paramètres suffixes uniquement comme des noms de variables ou de suffixes. Le meilleur exemple d'une macro vardef mettant en jeu des suffixes est la macro `z` qui définit la convention `z`. La définition met en jeu le *token* spécial `@#` qui se réfère au suffixe qui suit le nom de la macro

```
vardef z@#=(x@#,y@#) enddef;
```

```
def getmid(suffix p) =
   pair p.mid[], p.off[], p.dir[];
  for i=0 upto 36 :
    p.dir[i] = dir(5*i);
   p.mid[i] + p.off[i] = directionpoint p.dir[i] of p;
   p.mid[i] - p.off[i] = directionpoint -p.dir[i] of p;
   endfor
enddef;

def joinup(suffix pt,d)(expr n) =
  begingroup
  save res, g; path res;
  res = pt[0]{d[0]};
  for i=1 upto n :
   g := if (pt[i]-pt[i-1]) dotprod d[i] < 0 : - fi 1;
   res := res{g*d[i-1]} ... {g*d[i]}pt[i];
  endfor
  res
  endgroup
enddef;

beginfig(45)
path p, q;
p = ((5,2) .. (3,4) ... (1,3) ... (-2,-3) ... (0,-5)
... (3,-4) ... (5,-3) ... cycle) scaled .3cm shifted
(0,5cm);
getmid(p);
draw p;
draw joinup(p.mid, P.dir, 36) .. cycle;
q = joinup(p.off, p.dir, 36);
draw q .. (q rotated 180) .. cycle;
drawoptions(dashed evenly);
for i = 0 upto 3 :
  draw p.mid[9i] - p.off[9i] .. p.mid[9i] + p.off[9i];
  draw -p.off[9i] .. p.off[9i];
endfor
endfig;
```

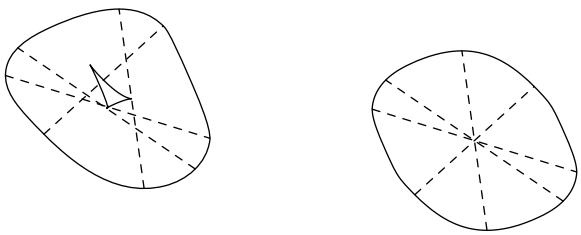


FIG. 45 – Code METAPOST et la figure correspondante (la figure a subi une rotation de $\pi/2$ en raison du format de la feuille)

Cela signifie que n'importe quel nom de variable dont le premier *token* (ou initiale) est équivalent au couple de variables dont les noms sont obtenus en remplaçant `z` par `x` et `y`. Par exemple, `z.a1` appelle la macro `z` avec le paramètre suffixe `@#` égal à `a1`.

En général,

`vardef`<variable générique>`@#`

est une alternative à `vardef`<variable générique> qui conduit l'interpréteur METAPOST à rechercher un suffixe suivant le nom donné dans l'appel de la macro et le rend disponible comme paramètre suffixe `@#`.

Pour résumer les caractéristiques spéciales des macros `vardef`, elles permettent une large classe de noms de macro aussi bien que des noms de macro suivis par un paramètre suffixe spécial. De plus, `begingroup` et `endgroup` sont ajoutés automatiquement au <texte de remplacement> d'une macro vardef. Ainsi l'utilisation de `vardef` au lieu de `def` pour définir la macro `joinup` dans la figure 45 aurait évité d'inclure explicitement `begingroup` et `endgroup` dans la définition de la macro.

En fait, la plupart des définitions de macro données dans les exemples précédents pourrait également utiliser `vardef` au lieu de `def`. Il est habituellement sans importance d'utiliser l'une ou l'autre forme, mais une bonne règle générale est d'utiliser `vardef` si l'on souhaite utiliser une macro comme une fonction ou une procédure. La comparaison suivante devrait aider dans la décision d'utiliser `vardef` :

- les macros `vardef` sont automatiquement encadrées par `begingroup` et `endgroup` ;
- la taille du nom d'une macro `vardef` peut être supérieure à un *token* et peut contenir des indices ;
- une macro `vardef` peut accéder au suffixe qui suit le nom d'une macro lors de l'appel de la macro ;
- lorsqu'un *token* symbolique est utilisé dans le nom d'une macro `vardef`, il demeure un *tag* et peut être encore utilisé dans d'autres noms de variables. Ainsi `p5dir` est un nom de variable légal même si `dir` est un nom d'une macro `vardef`, mais une macro ordinaire, telle que `...`, ne peut être employée dans un nom de variable (ceci est heureux puisque `z5...z6` est supposé être une expression de type chemin et non pas un nom de variable élaboré).

9.5. Définition des macros unaires et binaires

À plusieurs reprises, il a été signalé que la plupart des opérateurs et des commandes, qui ont été envisagés jusqu'à présent, sont en fait des macros prédéfinies. Elles englobent les opérateurs unaires tels que `round` et `unitvector`, les déclarations

telles que `fill` et `draw` et les opérateurs binaires comme `dotprod` et `intersectionpoint`. La différence majeure entre ces macros et celles déjà vues réside dans la façon de définir la syntaxe de leurs arguments.

Les macros `round` et `unitvector` sont des exemples de ce que la figure 14 appelle des <opérateurs unaires>. C'est-à-dire qu'elles sont suivies par une expression primaire. Pour spécifier un argument d'une macro de ce type, la définition de la macro doit ressembler à

`vardef round primary u =` <texte de remplacement> `enddef;`

Le paramètre `u` est un paramètre expression et peut être utilisé exactement comme le paramètre expression défini en utilisant la syntaxe habituelle

`(expr u)`

Comme le suggère l'exemple `round`, une macro peut être définie pour prendre un paramètre <secondaire>, <tertiaire> ou <expression>. Par exemple, la définition « prédéfinie » de la macro `fill` est grossièrement

```
def fill expr c = addto currentpicture contour c enddef;
```

Il est même possible de définir une macro pour jouer le rôle de l'<opérateur of> dans la figure 14. Par exemple, la macro `direction of` possède une définition de la forme

`vardef direction expr t of p =` <texte de remplacement> `enddef;`

Des macros peuvent aussi être définies pour se comporter comme des opérateurs binaires. Pour l'instant, la définition de la macro `dotprod` est de la forme :

`primarydef w dotprod z =` <texte de remplacement> `enddef;`

Cela transforme `dotprod` en <opérateur binaire primaire>. De même, `secondarydef` et `tertiarydef` introduisent la définition des <opérateurs binaires secondaires> et <opérateurs binaires tertiaires>. Toutes définissent des macros ordinaires, et non des macros `vardef` ; par exemple, il n'y a aucune « `primaryvardef` ».

Ainsi les définitions de macro peuvent être introduites par `def`, `vardef`, `primarydef`, `secondarydef` ou `tertiarydef`. Un <texte de remplacement> est n'importe quelle liste de *tokens* organisée en fonction du couple `def-enddef` où tous les cinq identificateurs de définition de macro sont traités comme `def` pour respecter la paire `def-enddef`.

Le reste de la syntaxe pour les définitions de macro est résumé figure 46. La syntaxe comporte quelques surprises. Les paramètres de macro peuvent avoir un <partie délimitée> et une <partie illimitée> (ou partie non délimitée). Normalement, l'une d'entre elles est <vide>, mais il est possible d'avoir les deux parties non vides à la fois :

```
def foo(text a) expr b =<texte de remplacement>enddef;
```

Cela définit une macro `foo` qui requiert un paramètre texte entre parenthèses suivi d'une expression.

La syntaxe permet également par la <partie non délimitée> de spécifier un argument de type `suffix` ou `text`. Un exemple de macro avec un paramètre suffixe non délimité est la macro prédéfinie `incr` qui est, en fait, définie comme ceci

```
vardef incr suffix $ = $ :=$+1; $ enddef;
```

Ceci transforme `incr` en une fonction qui requiert une variable, l'incrémente et renvoie la nouvelle valeur. Des paramètres suffixes non délimités peuvent être mis entre parenthèses, ainsi `incr a` et `incr (a)` sont toutes les deux correctes si `a` est une variable numérique. Il existe une macro prédéfinie similaire `decr` qui soustrait 1.

Les paramètres textes non délimités fonctionnent jusqu'à la fin d'une instruction. Plus précisément, un paramètre texte non délimité est une liste d'identificateurs suivant l'appel de la macro jusqu'au premier « `;` » ou « `endgroup` » ou « `end` » excepté qu'un argument contenant « `begingroup` » inclura toujours le « `endgroup` » correspondant.

Un exemple d'un paramètre texte illimité est donné avec la macro prédéfinie `cutdraw` dont la définition est plus ou moins celle-ci :

```
def cutdraw text t =
begingroup interim linecap :=butt; draw t; endgroup end-
def;
```

Cela rend la macro `cutdraw` synonyme de `draw` sauf en ce qui concerne la valeur de `linecap` (cette macro est fournie essentiellement pour assurer une compatibilité avec METAFONT).

<définition de macro>→ <en-tête de macro>=<texte de remplacement>`enddef`
<en-tête de macro>→ <*token* symbolique><partie délimitée><partie illimitée>
|`vardef`<variable générique><partie délimitée><partie illimitée>
|`vardef`<variable générique>`@#`<partie délimitée><partie illimitée>
| <def binaire>
<paramètre><*token* symbolique><paramètre><partie délimitée>→ <vide>
| <partie délimitée>`(`<type paramètre><paramètres *tokens*>`)`
<type paramètre> → `expr`|`suffix`|`text`
<paramètres *tokens*>→ <paramètre><paramètres *tokens*>,<paramètre>
<paramètre>→ <*tokens* symboliques>
<partie illimitée>→ <vide>
| <type paramètre><paramètre>
| <niveau de priorité><paramètre>
|`expr`<paramètre>`of`<paramètre>
<niveau de priorité>→ `primary`|`secondary`|`tertiary`
<def binaire>→ `primarydef`|`secondarydef`|`tertiarydef`

FIG. 46 – Syntaxe de définitions de macro

10. Boucles

De nombreux exemples des chapitres précédents ont utilisé des boucles `for` simples de la forme

`for`<*tokens* symboliques>`=`<expression>`upto` <expression> `:`
<texte de boucle>`endfor`

Il est également simple de construire une boucle qui compte en décroissant : il suffit de remplacer `upto` par `downto` en rendant la seconde <expression> inférieure à la première. Ce chapitre couvre des types plus compliqués de progressions, boucles dans lesquels le compteur de boucle se comporte comme un paramètre suffixe. Ce chapitre envisage également les sorties de boucle.

La première généralisation est suggérée par le fait que `upto` est une macro prédéfinie pour

```
step 1 until
```

et `downto` pour `step -1 until`. Un début de boucle

```
for i=a step b until c
```

balaye une séquence de valeurs `i` : `a`, `a+b`, `a+2b`,... s'arrêtant avant que `i` dépasse `c` ; c'est-à-dire que la boucle balaye les valeurs de `i` pour lesquelles `i`$\leq$`c` si `b`$>$`0` et `i`$\geq$`c` si `b`$<$`0`.

Il est préférable d'utiliser cette caractéristique seulement lorsque le pas est un entier ou un nombre qui peut être représenté exactement, en arithmétique à virgule fixe, comme un multiple de $\frac{1}{65536}$. Autrement, les erreurs vont s'accumuler et l'index de boucle peut ne pas atteindre la borne finale attendue. Par exemple,

```
for i=0 step .1 until 1 : show i; endfor
```

montre 10 valeurs de `i` dont la dernière est 0.90005.

La manière classique d'éviter les problèmes associés à un pas non entier est d'itérer sur des valeurs entières et ensuite de multiplier par un facteur d'échelle comme cela a été fait pour les figures 2 et 40.

D'une autre manière, les valeurs sur lesquelles on doit itérer peuvent être données explicitement. Toute séquence d'expressions (même vides) séparées par des virgules peut être utilisée à la place de `a step b upto c`. En fait, les expressions ne nécessitent pas d'être toutes du même type et ne nécessitent pas d'avoir des valeurs connues. Ainsi

```
for t=3.14, 2.78, (a,2a), "hello" : show a; endfor
```

montre les quatre valeurs énumérées.

On notera que le corps de la boucle dans l'exemple précédent est une instruction suivie par un point-virgule. Il est classique que le corps de boucle soit constitué d'une ou plusieurs instructions mais il ne s'agit pas d'une obligation. Une boucle est comme une définition de macro suivie d'un appel de macro. Le corps de la boucle peut être virtuellement n'importe quelle séquence de *tokens* du moment qu'ils aient un sens ensemble. Ainsi, l'instruction (ridicule)

```
draw for p=(3,1),(6,2),(7,5),(4,6),(1,3) : p -- endfor cycle;
```

est strictement équivalente à

```
draw (3,1) -- (6,2) -- (7,5) -- (4,6) -- (1,3) -- cycle;
```

La figure 18 présente un exemple plus réaliste d'une telle boucle.

Si une boucle ressemble à une définition de macro, l'index de boucle ressemble à un paramètre expression. Il peut représenter n'importe quelle valeur, mais ce n'est pas une

variable et ne peut pas être changé par une déclaration d'assignation. Pour réaliser ceci, il faut recourir à une boucle `forsuffixes`. Une boucle `forsuffixes` est presque comme une boucle `for` sauf que l'index de boucle se comporte comme un paramètre suffixe. La syntaxe est :

`forsuffixes`<*token* symbolique>`=` <liste de suffixes> `:`
<texte de boucle> `endfor`

où une <liste de suffixes> est une liste de suffixes séparés par des virgules. Si certains de ces suffixes sont <vides>, le <texte de boucle> sera exécuté avec le paramètre d'index mis à un suffixe vide.

Un bon exemple d'une boucle `forsuffixes` est la définition de la macro `dotlabels`

```
vardef dotlabels@#(text t)=
   forsuffixes $=t : dotlabel@#(str$,z$); endfor enddef;
```

Ceci explique pourquoi le paramètre de `dotlabels` doit être une liste de suffixes séparés par des virgules. La plupart des macros, qui acceptent des listes à longueurs variables, dont le séparateur est la virgule, utilisent celles-ci dans des boucles `for` ou `forsuffixes` de cette façon comme valeurs sur lesquelles la boucle itère.

S'il n'y a pas de valeurs sur lesquelles itérer, on peut utiliser une boucle `forever` :

`forever :` <texte de boucle> `endfor`

Pour mettre un terme à une telle boucle lorsqu'une condition booléenne est vérifiée, il faut utiliser une clause de sortie :

`exitif`<expression booléenne>`;`

Lorsque l'interpréteur METAPOST rencontre une clause de sortie, il évalue l'<expression booléenne> et sort de la boucle courante si l'expression est vraie. S'il est plus commode de sortir de la boucle quand l'expression est fausse, il faut alors utiliser la macro prédéfinie `exitunless`.

Ainsi la version METAPOST d'une boucle **while** est :

`forever : exitunless` <expression booléenne>`;`<texte de boucle>`endfor`

La clause de sortie peut tout aussi bien venir juste avant `endfor` ou n'importe où dans le <texte de boucle>. En fait, toute boucle `for`, `forever` ou `forsuffixes` peut contenir n'importe quel nombre de clauses de sortie.

Le résumé de la syntaxe des boucles (figure 47) ne fait pas mention explicitement des clauses de sortie parce qu'un <texte de boucle> peut être virtuellement n'importe quelle séquence de *tokens*. La seule restriction est qu'un <texte de boucle> doit être équilibré du point de vue des `for` et `endfor`. Naturellement, ce processus d'équilibre traite `forsuffixes` et `forever` exactement comme `for`.

<boucle>→ <en-tête de boucle> `:`<texte de boucle>`endfor`
<en-tête de boucle>→ `for`<*token* symbolique>`=`<progression>
|`for`<*token* symbolique>`=`<liste for>
|`forsuffixes`<*token* symbolique>`=`<liste de suffixes>
|`forever`
<progression>→ <expression numérique>`upto`<expression numérique>
| <expression numérique>`downto`<expression numérique>
| <expression numérique>`step`<expression numérique>`until`<expression numérique>
<liste for>→ <expression> | <liste for>`,`<expression>
<liste de suffixes>→ <suffixes> | <liste de suffixes>`,`<suffixes>

FIG. 47 – Syntaxe des boucles

11. Fabrication des boîtes

Cette partie décrit les macros auxiliaires non inclues dans METAPOST Plain qui permettent de réaliser facilement des choses faites pour être travaillées à partir de *pic*[3]. Ce qui suit décrit comment utiliser les macros contenues dans le fichier `boxes.mp`. Ce fichier est contenu dans un répertoire spécial réservé aux macros METAPOST et aux supports logiciels[8] et peut être atteint en donnant la commande METAPOST `input boxes` avant toute figure qui utilise les macros fabriquant les boîtes. La syntaxe pour la commande `input` est

`input` <nom de fichier>

dans laquelle l'extension « `.mp` » peut être omise. La commande `input` regarde d'abord dans le répertoire courant et ensuite dans le répertoire spécial des macros. Les utilisateurs intéressés par l'écriture des macros peuvent regarder le fichier `boxes.mp` dans ce répertoire.

8 Le nom de ce répertoire est souvent quelque chose du type `/usr/lib/mp/lib` mais ceci dépend du système.

11.1. Boîtes rectangulaires

L'idée principale des macros de fabrication de boîtes est qu'on doit dire

`boxit.`<suffixe>`(`<expression image>`)`

où le <suffixe> ne doit pas commencer par un indice[9]. Ceci crée des variables de type paire <suffixe>`.c`, <suffixe>`.n`, <suffixe>`.e`,... qui peuvent être utilisées pour le positionnement de l'image avant de la dessiner avec une commande séparée telle que

`drawboxed(`<liste de suffixes>`)`

L'argument de `drawboxed` doit être une liste de noms de boîtes séparées par des virgules où un nom de boîte est un <suffixe> avec lequel `boxit` a été appelé.

Pour la commande `boxit.bb(pic)`, le nom de boîte est `bb` et le contenu de la boîte est l'image `pic`. Dans ce cas, `bb.c` est la position du centre de la figure `pic` et `bb.sw`, `bb.se`, `bb.ne` et `bb.nw` sont les coins d'un chemin rectangulaire qui entoure la figure résultante. Les variables `bb.dx` et `bb.dy` donnent l'écart entre la version décalée de `pic` et le cadre rectangulaire, `bb.off` est la quantité dont `pic` a été décalée pour réaliser tout ceci.

Lorsque la macro `boxit` est appelée avec le nom de boîte b, elle donne les équations linéaires qui forcent b`.sw`, b`.se`, b`.ne` et b`.nw` à être les coins d'un rectangle aligné sur les axes x et y avec le contenu de la boîte centré à l'intérieur comme cela est indiqué par le rectangle gris de la figure 48. Les valeurs de b`.dx`, b`.dy` et b`.c` sont laissées indéterminées de manière à ce que l'utilisateur donne les équations pour positionner les boîtes. Si aucune de telles équations n'est donnée, des macros telles que `drawboxed` peuvent le détecter et donner des valeurs par défaut. Ces valeurs par défaut pour `dx` et `dy` sont contrôlées par les variables internes `defaultdx` et `defaultdy`.

Si b représente un nom de boîte, `drawboxed(`b`)` dessine la limite rectangulaire de la boîte b et ensuite le contenu de la boîte. Il est possible d'accéder séparément à cette frontière rectangulaire par `bpath b` ou en général par

`bpath`<nom de boîte>

[9] Certaines versions anciennes des macros de fabrication de boîtes ne permettaient aucun indice dans le suffixe de `boxit`.

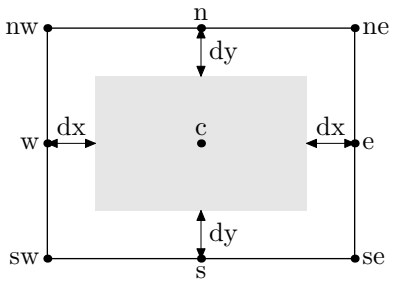


FIG. 48 – Relation entre la figure passée à `boxit` et les variables associées. La figure est matérialisée par le rectangle grisé

Cette commande est utile en conjonction avec des opérateurs tels que `cutbefore` et `cutafter` pour contrôler les chemins qui entrent dans une boîte. Par exemple, si *a* et *b* sont des noms de boîte et *p* un chemin depuis *a*`.c` jusqu'à *b*`.c`, l'instruction

`drawarrow` *p* `cutbefore bpath` *a* `cutafter bpath` *b*

dessine une flèche depuis le bord de la boîte *a* jusqu'au bord de la boîte *b*.

La figure 49 montre un exemple pratique incluant quelques flèches dessinées avec `cutafter bpath`<nom de boîte>. Il est instructif de comparer la figure 49 avec la figure similaire dans le manuel de *pic* [3]. La figure utilise une macro

`boxjoin(`<texte d'équation>`)`

pour contrôler la relation entre des boîtes consécutives. Dans le <texte d'équation>, `a` et `b` représentent les noms de boîte donnés dans les appels successifs de `boxit` et le <texte d'équation> donne les équations pour contrôler les tailles et positions relatives des boîtes.

Par exemple, la seconde ligne du fichier source de la figure précédente contient

```
boxjoin(a.se=b.sw; a.ne=b.nw)
```

Cela provoque l'alignement horizontal des boîtes en donnant des équations supplémentaires qui seront mises en jeu chaque fois qu'une boîte `a` est suivie par une autre boîte `b`. Ces équations sont d'abord traitées à la ligne suivante lorsque la boîte `a` est suivie par la boîte `ni`. Cela donne

```
a.se=ni.sw; a.ne=ni.nw
```

La paire suivante de boîtes est constituée par les boîtes `ni` et `di`. Cette fois, les équations implicitement produites sont

```
ni.se=di.sw; ni.ne=di.nw
```

Ce processus continue jusqu'à ce qu'une nouvelle `boxjoin` soit donnée. Dans ce cas la nouvelle déclaration est

```
boxjoin(a.sw=b.nw; a.se=b.ne)
```

indiquant que les boîtes doivent être empilées les unes sur les autres.

Après avoir appelé `boxit` pour les huit premières boîtes, depuis `a` jusqu'à `dk`, les hauteurs des boîtes sont ajustées mais les largeurs sont encore inconnues. Pour cela la macro `drawboxed` nécessite d'assigner des valeurs par défaut aux variables <nom de boîte>`.dx` et <nom de boîte>`.dy`. Tout d'abord, `di.dx` et `di.dy` prennent les valeurs par défaut de telle sorte que toutes les boîtes sont forcées d'être suffisamment larges pour recevoir le contenu de la boîte `di`.

La macro qui assigne réellement les valeurs par défaut aux variables `dx` et `dy` est appelée `fixsize`. Elle requiert une liste de noms de boîte et les considère une par une en s'assurant que chaque boîte a une taille et une forme fixée. Une macro appelée `fixpos` prend ensuite cette même liste de noms de boîte et assigne les valeurs par défaut à la variable <nom de boîte>`.off` nécessaire pour fixer la position de chaque boîte. En utilisant `fixsize` pour fixer les dimensions de chaque boîte avant d'assigner des positions par défaut à chacune d'elles, le nombre de positions par défaut peut être habituellement ramené à un seul.

Puisque le chemin de frontière pour une boîte ne peut être calculé tant que la taille, la forme et la position de la boîte ne sont pas déterminées, la macro `bpath` applique `fixsize` et `fixpos` à ses arguments. Les autres macros qui font cela incluent

`pic`<nom de boîte>

où le <nom de boîte> est un suffixe pouvant être entre parenthèses. Cela renvoie le contenu de la boîte nommée comme une figure positionnée de telle sorte que

`draw pic`<nom de boîte>

dessine le contenu de la boîte sans le rectangle de frontière. Cette opération peut être accomplie par la macro `drawunboxed` qui requiert une liste de noms de boîte séparés par des virgules. Il existe également la macro `drawboxes` qui trace seulement le rectangle frontière.

```
input boxes
beginfig(49);
boxjoin(a.se=b.sw; a.ne=b.nw);
boxit.a(btex\strut$ \cdots$ etex);
boxit.ni(btex\strut$n_i$ etex);
boxit.di(btex\strut$d_i$ etex);
boxit.ni1(btex\strut$n_{i+1}$ etex);
boxit.di1(btex\strut$d_{i+1}$ etex);
boxit.aa(btex\strut$ \cdots$ etex);
boxit.nk(btex\strut$n_k$ etex);
boxit.dk(btex\strut$d_k$ etex);
drawboxed(di,a,ni,ni1,di1,aa,nk,dk);
label.lft("ndtable :", a.w);
interim defaultdy :=7bp;
boxjoin(a.sw=b.nw; a.se=b.ne);
boxit.ba(); boxit.bb(); boxit.bc();
boxit.bd(btex $\vdots$ etex); boxit.be(); boxit.bf();
bd.dx=8bp; ba.ne=a.sw-(15bp,10bp);
drawboxed(ba,bb,bc,bd,be,bf); label.lft("hashtab :",ba.w);
vardef ndblock suffix $ =
  boxjoin(a.sw=b.nw; a.se=b.ne);
  forsuffixes $$=$1,$2,$3 : boxit$$(); ($$dx,$$dy)=(5.5bp,4bp);
  endfor; enddef;
ndblock nda; ndblock ndb; ndblock ndc;
nda1.c-bb.c = ndb1.c-nda3.c = (whatever,0);
xpart ndb3.se = xpart ndc1.ne = xpart di.c;
ndc1.c - be.c = (whatever,0);
drawboxed(nda1,nda2,nda3, ndb1,ndb2,ndb3, ndc1,ndc2,ndc3);
drawarrow bb.c -- nda1.w;
drawarrow be.c -- ndc1.w;
drawarrow nda3.c -- ndb1.w;
drawarrow nda1.c{right}..{curl0}ni.c cutafter bpath ni;
drawarrow nda2.c{right}..{curl0}di.c cutafter bpath di;
drawarrow ndc1.c{right}..{curl0}ni1.c cutafter bpath ni1;
drawarrow ndc2.c{right}..{curl0}di1.c cutafter bpath di1;
drawarrow ndb1.c{right}..nk.c cutafter bpath nk;
drawarrow ndb2.c{right}..dk.c cutafter bpath dk;
x.ptr=xpart aa.c; y.ptr=ypart ndc1.ne;
drawarrow subpath (0,.7) of (z.ptr..{left}ndc3.c) dashed
evenly;
label.rt(btex \strut ndblock etex, z.ptr); endfig;
```

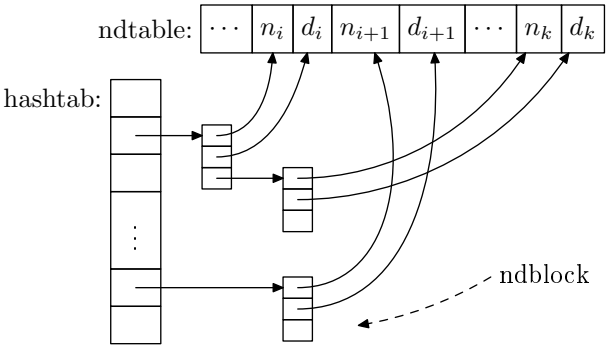


FIG. 49 – Code METAPOST et la figure résultante

Une autre manière de dessiner un rectangle vide peut se faire en disant juste

`boxit`<nom de boîte>`()`

sans argument de type figure ou dessin. Cela est effectué plusieurs fois dans la figure 49. C'est comme si l'on appelait `boxit` avec une figure vide. D'autre part, l'argument peut également être une expression chaîne de caractères au lieu d'une expression dessin. Dans ce cas la chaîne sera affichée avec la fonte courante.

11.2. Boîtes circulaires et ovales

Les boîtes circulaires et ovales sont très semblables aux boîtes rectangulaires mise à part la forme du chemin frontière. De telles boîtes sont mises en place grâce à la macro `circleit` :

`circleit`<nom de boîte>`(`<contenu de boîte>`)`

où le <nom de boîte> est un suffixe et <contenu de boîte> une expression figure, une expression chaîne ou une expression <vide>.

La macro `cercleit` définit des variables de type paire tout comme le fait `boxit`, sauf qu'il n'y a pas de point de coin <nom de boîte>`.ne`, <nom de boîte>`.sw`, etc. Un appel à

`cercleit.a(...)`

donne des relations entre les points `a.c`, `a.s`, `a.e`, `a.n`, `a.w` et les distances `a.dx` et `a.dy`. Avec `a.c` et `a.off`, ces variables décrivent la manière dont la figure est centrée dans un ovale comme on peut le voir sur la figure 50.

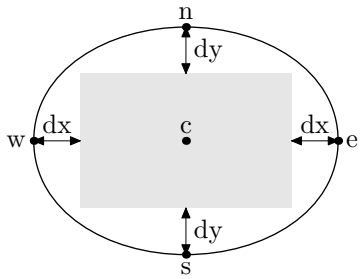


FIG. 50 – Relation entre la figure passée à `cercleit` et les variables associées. La figure est matérialisée par le rectangle grisé

Les macros `drawboxed`, `drawunboxed`, `drawboxes`, `pic`, et `bpath` fonctionnent pour les boîtes `cercleit` comme elles fonctionnaient pour les boîtes `boxit`. Par défaut, le chemin frontière d'une boîte `cercleit` est un cercle suffisamment grand pour entourer le contenu de la boîte avec une petite marge de sécurité contrôlée par la variable interne `circmargin`. La figure 51 donne un exemple élémentaire de l'utilisation de `bpath` avec une boîte `cercleit`.

```
vardef drawshadowed(text t) =
  fixsize(t);
  forsuffixes s=t :
   fill bpath.s shifted (1pt,-
1pt);
   unfill bpath.s;
   drawboxed(s);
  endfor
enddef;

beginfig(51)
circleit.a(btex Box 1 etex);
circleit.b(btex Box 2 etex);
b.n = a.s - (0,20pt);
drawshadowed(a,b);
drawarrow a.s -- b.n;
endfig;
```

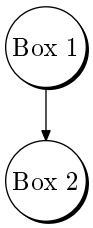


FIG. 51 – Code METAPOST et la figure résultante. Il faut noter que la macro `drawshadowed` utilisée ici ne fait pas partie du package de macros `boxes.mp`

```
vardef cuta(suffix a,b) expr p =
  drawarrow p cutbefore bpath.a cutafter bpath.b;
  point .5*length p of p
enddef;
vardef self@# expr p =
  cuta(@#,@#) @#.c{curl0}..@#.c+p..{curl0}@#.c  enddef;


beginfig(52); verbatimtex
\def\stk#1#2{$\displaystyle{\matrix{#1\cr#2\cr}}$} etex
circleit.aa(btex\strut Start etex); aa.dx=aa.dy;
circleit.bb(btex \stk B{(a|b)^*a} etex);
circleit.cc(btex \stk C{b^*} etex);
circleit.dd(btex \stk D{(a|b)^*ab} etex);
circleit.ee(btex \strut Stop etex); ee.dx=ee.dy;
numeric hsep; bb.c-aa.c = dd.c-bb.c = ee.c-dd.c = (hsep,0);
cc.c-bb.c = (0,.8hsep);
xpart(ee.e - aa.w) = 3.8in;
drawboxed(aa,bb,cc,dd,ee);
label.ulft(btex $b$ etex, cuta(aa,cc) aa.c{dir50} .. cc.c);
label.top(btex $b$ etex, self.cc(0,30pt));
label.rt(btex $a$ etex, cuta(cc,bb) cc.c .. bb.c);
label.top(btex $a$ etex, cuta(aa,bb) aa.c .. bb.c);
label.llft(btex $a$ etex, self.bb(-20pt,-35pt));
label.top(btex $b$ etex, cuta(bb,dd) bb.c .. dd.c);
label.top(btex $b$ etex, cuta(dd,ee) dd.c .. ee.c);
label.lrt(btex $a$ etex, cuta(dd,bb) dd.c .. {dir140}bb.c);
label.bot(btex $a$ etex, cuta(ee,bb) ee.c .. tension1.3 ..
{dir115}bb.c);
label.urt(btex $b$ etex, cuta(ee,cc) ee.c{(cc.c-ee.c)
rotated-15} .. cc.c);
endfig;
```

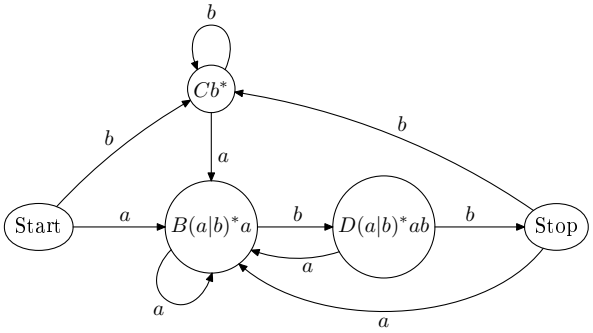


FIG. 52 – Code METAPOST et la figure résultante

Un exemple complet de boîtes `cercleit` apparaît dans la figure 52. Les frontières ovales autour de « Start » et « Stop » proviennent des équations `aa.dx=aa.dy;` et `ee.dx=ee.dy;` après `circleit.ee(btex\strut Start etex)` et `circleit.ee(btex\strut Stop etex)`.

La règle générale est que `bpath.`*c* est circulaire si *c*`.dx`, *c*`.dy` et *c*`.dx`-*c*`.dy` sont tous inconnus. Sinon les macros sélectionnent un ovale suffisamment grand pour contenir la figure donnée avec un marge de sécurité `circmargin`.

12. Déboguage

METAPOST hérite de METAFONT de nombreuses facilités de déboguage interactif dont les caractéristiques ne seront que brièvement mentionnées dans ce chapitre. On trouvera plus d'informations sur les messages d'erreur, le déboguage et l'affichage d'information dans le METAFONTbook [4].

Supposons que le fichier source indique

```
draw z1 -- z2;
```

sur la ligne 17 sans avoir donné au préalable des valeurs connues pour `z1` et `z2`. La figure 53 montre ce que l'interpréteur METAPOST affiche sur l'écran quand il trouve l'erreur. Le message d'erreur effectif est la ligne qui commence par « `!` » ; les six lignes suivantes indiquent le contexte qui montre exactement quelle entrée a été lue lorsque l'erreur a été détectée et le « `?` » de la dernière ligne est une invite pour la réponse de l'utilisateur. Comme le message d'erreur indique une abscisse inconnue, cette valeur est affichée sur la première ligne, après le « >> ». Dans ce cas, l'abscisse de `z1` est justement la variable `x1` inconnue, l'interpréteur affiche donc le nom de variable `x1` exactement comme s'il avait été demandé « `show x1` » à ce niveau.

```
>> x1
! Undefined x coordinate has been replaced by 0.
<to be read again>
                   {
-- ->{
       curl1}..{curl1}
1.17 draw z1 --
             z2;
?
```

FIG. 53 – Exemple de message d'erreur

Le listing du contexte ainsi affiché peut paraître un peu confus au début. En fait, il donne simplement quelques lignes de textes montrant comment chaque ligne a été lue jusqu'ici.

Chaque ligne du source est affichée sur deux lignes de la façon suivante

```
<descripteur$>$ Texte lu jusqu'ici
                                Texte qui va être lu
```

Le <descripteur> identifie le fichier source. Il s'agit soit d'un numéro de ligne comme « `1.17` » pour la ligne 17 du fichier courant, soit un nom de macro suivi par « `->` » soit enfin une phrase descriptive entre crochets. Ainsi, le listing de contexte de la figure 53 indique que l'interpréteur a lu la ligne 17 du fichier source jusqu'à « `--` » , que le développement de la macro `--` a commencé et que le « `{` » initial a été réinséré pour permettre à l'utilisateur d'insérer quelque chose avec ce *token*.

Parmi les réponses possibles au prompt `?` on trouve :

x termine la compilation pour corriger le fichier source et à relancer une compilation ;
h affiche un message d'aide suivi par un autre prompt `?` ;
<**return**> force le compilateur à continuer comme il peut ;
? affiche une liste d'options disponibles, suivie par un autre `?`.

Les messages d'erreur et les réponses aux commandes `show` sont également écrits dans le fichier de transcription dont le nom est obtenu à partir du nom du fichier source principal en changeant « `.mp` » en « `.log` ». Lorsque la variable interne `tracingonline` a sa valeur par défaut, qui est nulle, certaines commandes `show` écrivent leurs résultats de manière plus détaillée seulement dans le fichier de transcription.

Un seul type de commande `show` a été étudié jusqu'ici : `show` suivi par une liste d'expressions séparées par des virgules qui affichait des représentations symboliques de ces expressions.

La commande `showtoken` peut être utilisée pour montrer les paramètres et le texte de remplacement d'une macro. Cette commande accepte une liste de *tokens* séparés par des virgules et identifie chacun d'eux. Si le *token* est une primitive, comme pour « `showtoken +` », elle est uniquement identifiée comme étant elle-même

```
> +=+
```

En appliquant `showtoken` à une variable ou une macro `vardef` on obtient

```
> <token>=variable
```

Pour plus d'information sur une variable, il faut utiliser `showvariable` à la place de `showtoken`. L'argument de `showvariable` est une liste de *tokens* symboliques séparés par des virgules et le résultat une description de toutes les variables dont le nom commence par l'un des *tokens* de la liste. Cela fonctionne même pour les macros `vardef`. Par exemple, `showvariable z` donne

```
z@#=macro :->begingroup(x(SUFFIX2),y(SUFFIX2))endgroup
```

Il existe également une commande `showdependencies` qui ne requiert aucun argument et affiche la liste de toutes les variables *dépendantes* et comment les équations linéaires données jusqu'ici les rendent dépendantes d'autres variables. Ainsi, après

```
z2-z1=(5,10); z1+z2=(a,b);
```

`showdependencies` affiche le contenu de la figure 54. Cela peut être très utile pour répondre à une question du genre « Que signifie `'! Undefined x coordonate?'` *(Coordonnée x indéfinie)* Je pensais que les équations suivantes auraient servi à déterminer `x1`. »

```
x2=0.5a+2.5
y2=0.2b+5
x1=0.5a-2.5
y1=0.5b-5
```

FIG. 54 – Le résultat de `z2-z1=(5,10); z1+z2=(a,b); showdependencies;`

Quand tout va mal, la macro `tracingall` indique à l'interpréteur d'afficher un listing détaillé de tout ce qu'il fait. Comme les informations sont alors souvent assez volumineuses, il peut être préférable d'utiliser la macro `logginall` qui produit exactement les mêmes informations mais ne fait que les écrire dans le fichier de transcription. Il y a également une macro `tracingnone` qui inhibe toutes les sorties de déboguage.

Les sorties de déboguage sont contrôlées par un ensemble de variables internes résumées ci-dessous. Lorsqu'une de ces variables contient une valeur strictement positive, la forme correspondante de sortie est activée. Voici cet ensemble de variables de contrôle de sortie et ce qui arrive lorsque leur contenu devient strictement positif :

`tracingcapsules` montre les valeurs des quantités temporaires (capsules) lorsqu'elles deviennent connues ;

`tracingchoices` montre les points de contrôle de Bézier de chaque nouveau chemin lorsqu'il est choisi ;

`tracingcommands` montre les commandes avant qu'elles soient effectuées. Une initialisation > 1 montre également les tests `if` et les boucles avant qu'ils ne soient développés et une initialisation > 2 montre les opérations algébriques avant qu'elles ne soient effectuées ;

`tracingequations` montre chaque variable lorsqu'elle devient connue ;

`tracinglostchars` avertit au sujet de caractères omis en raison de leur absence dans la fonte courante des légendes ;

`tracingmacros` montre les macros avant qu'elles ne soient développées ;

`tracingoutput` montre les dessins comme ils ont été envoyés dans les fichiers POSTSCRIPT ;

tracingrestores montre les symboles et les variables internes tels qu'ils sont restaurés à la fin d'un groupe ;
tracingspecs montre les contours générés lors d'un tracé avec un stylo polygonal ;
tracingstats écrit dans le fichier de transcription, à la fin du travail, la quantité de ressources utilisée par l'interpréteur METAPOST.

13. Remerciements

Je voudrais remercier Don KNUTH pour avoir rendu ce travail possible en ayant développé METAFONT et en le plaçant dans le domaine public. Je lui suis également reconnaissant de ses différentes suggestions fort utiles, particulièrement en ce qui concerne le traitement des matériels TEX inclus.

Les traducteurs s'associent pleinement à ces remerciements. Ils remercient l'auteur d'avoir permis la diffusion de cette version française dans les *Cahiers* GUTENBERG.

A. Manuel de référence

Les tableaux 3 – 11 résument les fonctionnalités programmées dans Plain METAPOST et les caractéristiques définies dans le fichier de macros `boxes.mp`. Comme il est expliqué au chapitre 11, le fichier `boxes.mp` n'est pas automatiquement chargé et les macros définies ne seront pas accessibles tant qu'elles n'auront pas été demandées par la commande

```
input boxes
```

Les caractéristiques qui dépendent de `boxes.mp` sont marquées par des symboles ‡. Celles qui dépendent du package de macros Plain sont marquées avec des symboles † et les primitives de METAPOST ne sont pas marquées. La distinction entre primitives et macros de Plain peut être ignorée par un simple utilisateur mais il est important de se souvenir que les caractéristiques marquées d'un ‡ ne peuvent être lues qu'après avoir chargé le fichier de macros `boxes.mp`.

Les tableaux de cet appendice donnent le nom de chaque caractéristique, le numéro de page où elle est présentée ainsi qu'une courte description. Quelques caractéristiques n'ont jamais été présentées et n'ont donc pas de numéro de page associé. Ces caractéristiques existent principalement pour des raisons de compatibilité avec METAFONT et ne sont pas prévues pour être explorées pour elles-mêmes. Certaines autres caractéristiques de METAPOST sont omises complètement car elles ne présentent qu'un intérêt limité ou nécessiteraient de très longues explications. Toutes ces caractéristiques sont documentées dans le METAFONT*book* [4] comme l'explique l'appendice B.

Le tableau 3 donne la liste des variables internes qui prennent des valeurs numériques en argument. La table 4 donne la liste des variables prédéfinies des autres types. Le tableau 5 donne la liste des constantes prédéfinies. Certaines de celles-ci sont implémentées en tant que variables dont les valeurs sont supposées rester inchangées.

Les tableaux 6 à 9 résument les opérateurs METAPOST et donnent la liste des types d'arguments et de sortie possibles pour chacun d'eux. Une entrée « – » pour l'argument gauche indique un opérateur unaire ; « – » pour les deux arguments indique un opérateur isolé. Les opérateurs qui prennent des paramètres suffixes ne sont pas listés dans ces tableaux car ils sont traités comme des « macros-fonctions ».

Les deux derniers tableaux sont le tableau 10 pour les commandes et le tableau 11 pour les macros qui se comportent comme des fonctions ou des procédures. De telles macros prennent comme arguments des listes entre parenthèses ou des paramètres suffixes et retournent soit une valeur dont le type est listé dans la table, soit rien du tout. Ce dernier cas se présente pour les macros qui se comportent comme des procédures ; la valeur de retour est alors mentionnée sous la forme « – ».

Les figures dans cet appendice présentent la syntaxe du langage METAPOST en commençant par les expressions en figures 55 à 57. Bien que les productions spécifient quelquefois des types pour les expressions, les primaires, les secondaires et les tertiaires, aucun effort n'a été fait pour donner des syntaxes séparées pour les <expressions numériques>, <expression paire>, etc. La simplicité des productions de la figure 58 est due à ce manque d'informations du type. Les informations sur les types pourront être trouvées dans les tableaux 3 à 11.

Les figures 59 et 60 donnent la syntaxe pour les programmes METAPOST, y compris celles des instructions et des commandes. Elles ne mentionnent pas les boucles et les tests `if` car ces constructions ne se comportent pas comme des instructions. La syntaxe donnée dans les figures 55 à 57 s'applique aux résultats du développement de toutes les boucles et de tous les tests conditionnels. Les conditions et les boucles ont une syntaxe mais celle-ci se compose de séquences de *tokens* presque arbitraires. La figure 61 spécifie les conditions, en termes de <*tokens* équilibrés>, et les boucles, en tant que <texte de boucle> où les <*tokens* équilibrés> sont n'importe quelle séquence de *tokens* équilibrés vis-à-vis des `if` et des `fi`, et où <texte de boucle> indique une séquence de *tokens* équilibrés vis à vis des `for`, `forsuffixes`, `forever` et `endfor`.

TAB. 3: Variables internes avec leur valeur numérique

Nom	**Page**	**Explication**
†ahangle	69	angle (en degrés) des têtes de flèches [45]
†ahlength	69	taille des têtes de flèche [4bp]
†bboxmargin	50	espace supplémentaire permis par bbox [2bp]
charcode	75	numéro du prochain caractère devant être sorti
‡circmargin	100	zone libre autour du contenu d'une boîte circulaire ou ovale
day	--	le jour courant du mois
‡defaultdx	95	espace libre horizontal par défaut autour du contenu d'une boîte [3bp]
‡defaultdy	95	espace libre vertical par défaut autour du contenu d'une boîte [3bp]
†defaultpen	72	index numérique utilisé par pickup pour sélectionner le stylo par défaut
†defaultscale	46	facteur d'agrandissement de fonte pour les légendes [1]
†labeloffset	44	éloignement des légendes [3bp]
linecap	66	0 pour butt, 1 pour round, 2 pour square
linejoin	67	0 pour mitered, 1 pour round, 2 pour beveled
miterlimit	68	contrôle la longueur de raccordement comme en POSTSCRIPT
month	--	le mois courant (par exemple, 3 pour mars)
pausing	--	> 0 affiche les lignes sur le terminal avant qu'elles soient lues
prologues	49	> 0 rend la sortie conforme aux fontes définies en POSTSCRIPT
showstopping	--	> 0 pour stopper après chaque commande show
time	--	le nombre de minutes écoulées depuis minuit lorsque la compilation a commencé
tracingcapsules	104	> 0 montre les capsules
tracingchoices	104	> 0 montre les points de contrôles choisis pour les chemins
tracingcommands	104	> 0 montre les commandes et les opérations alors qu'elles sont exécutées
tracingequations	104	> 0 montre chaque variable lorsqu'elle devient connue
tracinglostchars	104	> 0 montre les caractères absents de la fonte

TAB. 3: Variables internes avec leur valeur numérique (suite)

Nom	**Page**	**Explication**
`tracingmacros`	104	> 0 montre les macros avant qu'elles soient développées
`tracingonline`	33	> 0 montre des informations longues sur le terminal
`tracingoutput`	104	> 0 montre les chemins digitalisés tels qu'ils sont sortis
`tracingrestores`	105	> 0 montre les moments où une variable externe ou interne est restaurée
`tracingspecs`	105	> 0 montre les subdivisions de chemin lorsqu'un stylo polygonal est utilisé
`tracingstats`	105	> 0 montre l'utilisation de la mémoire à la fin de la compilation
`tracingtitles`	--	> 0 montre les titres en ligne lorsqu'ils apparaissent
`truecorners`	51	> 0 pour que `llcorner`, ... ignorent `setbounds`
`warningcheck`	33	contrôle les messages d'erreur lorsqu'une valeur de variable devient trop grande
`year`	--	l'année courante (par exemple 2000)

TAB. 4: Autres variables prédéfinies

Nom	**Type**	**Page**	**Explication**
†`background`	couleur	53	couleur pour `unfill` et `undraw` [white]
†`currentpen`	stylo	71	dernier stylo choisi (pour une utilisation avec `draw`)
†`currentpicture`	dessin	35	accumulation des résultats des commandes `draw` et `fill`
†`cuttings`	chemin	58	dernier sous-chemin coupé par `cutbefore` ou `cutafter`
†`defaultfont`	chaîne	45	fonte utilisée pour les légendes avec du texte
†`extra_beginfig`	chaîne	131	commandes que `beginfig` doit exécuter
†`extra_endfig`	chaîne	131	commandes que `endfig` doit exécuter

TAB. 5: Constantes prédéfinies

Nom	Type	Page	Explication
†beveled	numérique	67	valeur de `linejoin` pour les jonctions biseautées
†black	couleur	34	équivalent de `(0,0,0)`
†blue	couleur	34	équivalent de `(0,0,1)`
†bp	numérique	20	un point POSTSCRIPT en unités bp [1]
†butt	numérique	67	valeur de `linecap` pour les terminaisons butt [0]
†cc	numérique	--	un cicéro en unités bp [12.79213]
†cm	numérique	20	un centimètre en unités bp [28.34645]
†dd	numérique	--	un didot en unités bp [1.06601]
†ditto	chaîne	--	le caractère " en tant que chaîne de longueur 1
†down	paire	25	vecteur de direction vers le bas (0,-1)
†epsilon	numérique	--	le plus petit nombre strictement positif en METAPOST [$\frac{1}{65536}$]
†evenly	dessin	63	motif de points avec des tirets d'égale longueur
false	booléen	35	la valeur booléenne fausse
†fullcircle	chemin	52	cercle de diamètre 1 centré en (0,0)
†green	couleur	34	équivalent de `(0,1,0)`
†halfcircle	chemin	53	demi-cercle supérieur d'un cercle de diamètre 1
†identity	transformation	61	transformation identité
†in	numérique	20	un pouce en unités bp [72]
†infinity	numérique	57	valeur positive la plus grande [4095.99998]
†left	paire	25	direction gauche (-1,0)
†mitered	numérique	67	valeur de `linejoin` pour les jonction avec raccordement
†mm	numérique	20	un millimètre en unités bp [2.83464]
nullpicture	dessin	37	dessin vide
origin	paire	--	la paire (0,0)
†pc	numérique	--	un pica en unités bp [11.95517]
pencircle	stylo	72	stylo circulaire de diamètre 1
†pensquare	stylo	72	stylo carré de hauteur et de largeur 1
†pt	numérique	20	un point imprimante en unités bp [0.99626]

TAB. 5: Constantes prédéfinies (suite)

Nom	**Type**	**Page**	**Explication**
†`quartercircle`	chemin	--	premier quart d'un cercle de diamètre 1
†`red`	couleur	34	équivalent à `(1,0,0)`
†`right`	paire	25	direction vers la droite (1,0)
†`rounded`	numérique	67	valeur de `linejoin` et de `linecap` pour les connexions arrondies et les terminaisons arrondies [1]
†`squared`	numérique	67	valeur de `linecap` pour les terminaisons carrées [2]
`true`	booléen	35	la valeur booléenne vraie
†`unitsquare`	chemin	--	le chemin `(0,0) -- (1,0) -- (1,1) -- (0,1) -- cycle`
†`up`	paire	25	direction vers le haut (0,1)
†`white`	couleur	34	équivalent à `(1,1,1)`
†`withdots`	dessin	63	motifs de points produisant des lignes en pointillés

TAB. 6: Opérateurs (première partie)

Nom	Types arguments/sortie			Page	Explication
	Gauche	**Droite**	**Résultat**		
&	chaîne chemin	chaîne chemin	chaîne chemin	36	concaténation : pour fonctionner, il faut qu'en cas de $g\&d$, d débute exactement là où g se termine
*	numérique	couleur numérique paire	couleur numérique paire	36	multiplication
*	couleur numérique paire	numérique	couleur couleur paire	36	multiplication
**	numérique	numérique	numérique	36	exponentiation
+	couleur numérique paire	couleur numérique paire	couleur numérique paire	36	addition
++	numérique	numérique	numérique	36	addition pythagoricienne $\sqrt{g^2+r^2}$
+-+	numérique	numérique	numérique	36	soustraction pythagoricienne $\sqrt{g^2-d^2}$
-	couleur numérique paire	couleur numérique paire	couleur numérique paire	36	soustraction
-	--	couleur numérique paire	couleur numérique paire	36	négation
/	couleur numérique paire	numérique	couleur numérique paire	36	division

TAB. 6: Opérateurs (première partie) (suite)

Nom	Types arguments/sortie			Page	Explication
	Gauche	**Droite**	**Résultat**		
<=> <= >= <>	chaîne numérique paire couleur transformation	chaîne numérique paire couleur transformation	booléen	35	opérateurs de comparaison
†abs	–	numérique paire	numérique	39	valeur absolue
and	booléen	booléen	booléen	35	et logique
angle	–	paire	numérique	39	arctangente des deux arguments (en degrés)
arclength	–	chemin	numérique	60	longueur d'arc d'un chemin
arctime of	numérique	chemin	numérique	81	temps du chemin où la longueur de l'arc à partir du début atteint la valeur donnée en argument
ASCII	–	chaîne	numérique	–	valeur ASCII du premier caractère de la chaîne
†bbox	–	dessin chemin plume	chemin	50	*bounding box*
bluepart	–	couleur	numérique	40	extrait la troisième composante
boolean	–	quelconque	booléen	39	l'expression est-elle booléenne ?
bot	–	numérique paire	numérique paire	72	bas du stylo courant lorsqu'il est centré sur les coordonnées données en argument
†ceiling	–	numérique	numérique	39	plus petit entier supérieur ou égal à
†center	–	dessin chemin plume	paire	50	centre de la *bounding box*

TAB. 7: Opérateurs (deuxième partie)

Nom	Types arguments/sortie			Page	Explication
	Gauche	**Droite**	**Résultat**		
`char`	–	numérique	chaîne	49	caractère ayant le code ASCII spécifié
`color`	–	quelconque	booléen	39	l'expression est-elle de type couleur ?
`cosd`	–	numérique	numérique	39	cosinus de l'angle en degrés
†`cutafter`	chemin	chemin	chemin	58	partie de l'argument gauche avant l'intersection avec l'argument droit
†`cutbefore`	chemin	chemin	chemin	57	partie de l'argument gauche après l'intersection avec l'argument droit
`cycle`	–	chemin	booléen	39	détermine si le chemin est un cycle ou non
`decimal`	–	numérique	chaîne	39	représentation décimale
†`dir`	–	numérique	paire	25	$(\cos\theta, \sin\theta)$ où θ est donné en degrés
†`directionof`	numérique	chemin	paire	58	direction d'un chemin au « temps » donné
†`directionpointof`	paire	chemin	numérique	60	point où le chemin a la direction donnée
`directiontime of`	paire	chemin	numérique	58	« temps » où le chemin a la direction donnée
†`div`	numérique	numérique	numérique	–	division entière $\lfloor l/r \rfloor$
†`dotprod`	paire	paire	numérique	36	produit scalaire de deux vecteurs
`floor`	–	numérique	numérique	39	plus grand entier inférieur ou égal à
`fontsize`	–	chaîne	numérique	46	la taille du corps d'une fonte
`greenpart`	–	couleur	numérique	40	extrait la seconde composante
`hex`	–	chaîne	numérique	–	interprète l'argument comme un nombre hexadécimal

TAB. 7: Opérateurs (deuxième partie) (suite)

Nom	Types arguments/sortie			Page	Explication
	Gauche	**Droite**	**Résultat**		
`infont`	chaîne	chaîne	dessin	49	imprime la chaîne dans la fonte donnée
†`intersectionpoint`	chemin	chemin	paire	55	point d'intersection
`intersectiontimes`	chemin	chemin	paire	56	temps (t_g, t_d) sur les chemins g et d lorsqu'ils se coupent
†`inverse`	–	transformation	transformation	62	inverse une transformation
`known`	–	quelconque	booléen	39	l'argument a-t-il une valeur connue ?
`length`	–	chemin	numérique	57	nombre d'arcs dans un chemin
†`lft`	–	numérique	numérique	44	côté gauche d'un stylo lorsqu'il est centré au niveau de la(les) coordonnée(s) donnée(s)
`llcorner`	–	dessin chemin stylo	paire	50	coin inférieur gauche d'une *bounding box*
`lrcorner`	–	dessin chemin stylo	paire	50	coin inférieur droit d'une *bounding box*
`makepath`	–	stylo	chemin	72	chemin cyclique bordant la forme du stylo
`makepen`	–	chemin	stylo	72	stylo polygonal construit à partir de l'enveloppe convexe des sommets du chemin donné
`mexp`	–	numérique	numérique	–	la fonction $\exp(x/256)$
`mlog`	–	numérique	numérique	–	la fonction $256\ln(x)$
†`mod`	–	numérique	numérique	–	la fonction reste $g - d\lfloor g/d \rfloor$
`normaldeviate`	–	–	numérique	–	choix d'un nombre aléatoire de moyenne nulle et d'écart-type égal à 1

TAB. 8: Opérateurs (troisième partie)

Nom	Types arguments/sortie			Page	Explication
	Gauche	**Droite**	**Résultat**		
`not`	–	booléen	booléen	35	négation logique
`numeric`	–	quelconque	booléen	39	l'argument est-il numérique ?
`oct`	–	chaîne	numérique	–	interprète une chaîne comme un nombre octal
`odd`	–	numérique	booléen	–	l'argument est-il impair ?
`or`	booléen	booléen	booléen	35	ou inclusif logique
`pair`	–	quelconque	booléen	33	l'argument est-il une paire ?
`path`	–	quelconque	booléen	33	l'argument est-il un chemin ?
`pen`	–	quelconque	booléen	33	l'argument est-il un stylo ?
`penoffset of`	paire	stylo	paire	–	point du stylo le plus éloigné dans la direction donnée
`picture`	–	quelconque	booléen	33	l'argument est-il un dessin ?
`point of`	numérique	chemin	paire	57	point du chemin au temps donné
`postcontrol of`	numérique	chemin	paire	–	premier point de contrôle de BÉZIER sur le segment du chemin débutant au temps donné
`precontrol of`	numérique	chemin	paire	–	dernier point de contrôle de BÉZIER sur le segment du chemin débutant au temps donné
`redpart`	–	couleur	numérique	40	extrait la première composante
`reverse`	–	chemin	chemin	69	chemin à « temps » inversé (échange le début avec la fin)
`rotated`	dessin chemin paire stylo transformation	numérique	dessin chemin paire stylo transformation	60	rotation d'un angle donné en degrés dans le sens trigonométriques

TAB. 8: Opérateurs (troisième partie) (suite)

Nom	Types arguments/sortie			Page	Explication
	Gauche	**Droite**	**Résultat**		
†round	–	numérique paire	numérique paire	39	arrondit chaque composant à l'entier le plus proche
†rt	–	numérique paire	numérique paire	72	bord droit du stylo courant lorsqu'il est centré sur la(les) coordonnée(s) spécifiée(s)
scaled	dessin chemin paire stylo transformation	numérique	dessin chemin paire stylo transformation	60	multiplie toutes les coordonnées par la quantité donnée
shifted	dessin chemin paire stylo transformation	numérique	dessin chemin paire stylo transformation	60	ajoute à chaque coordonnée la quantité mentionnée
sind	–	numérique	numérique	39	sinus d'un angle donné en degrés
slanted	dessin chemin paire stylo transformation	numérique	dessin chemin paire stylo transformation	60	applique la transformation d'inclinaison qui transforme (x, y) en $(x + sy, y)$ où s est l'argument numérique
sqrt	–	numérique	numérique	39	racine carrée
str	–	suffixe	chaîne	86	représentation sous forme d'une chaîne d'un suffixe

TAB. 9: Opérateurs (quatrième partie)

Nom	Types arguments/sortie			Page	Explication
	Gauche	**Droite**	**Résultat**		
`string`	–	quelconque	booléen	39	l'argument est-il une chaîne ?
`subpath of`	paire	chemin	chemin	57	portion d'un chemin compris entre deux valeurs de temps
`substring of`	paire	chaîne	chaîne	37	sous-chaîne délimitée par les indices donnés
†`top`	–	numérique paire	numérique paire	72	haut du stylo courant lorsqu'il est centré sur la (les) coordonnée(s) donnée(s)
`transform`	–	quelconque	booléen	39	l'argument est-il une transformation ?
`transformed`	dessin chemin paire stylo transformation	transformation	dessin chemin paire stylo transformation	62	applique la transformation courante sur toutes les coordonnées
`ulcorner`	–	dessin chemin	paire	50	coin supérieur gauche de la *bounding box*
`uniformdeviate`	–	numérique	numérique	–	nombre aléatoire compris entre zéro et la valeur de l'argument
†`unitvector`	–	paire	paire	39	vecteur normé de même direction que l'argument
`unknown`	–	quelconque	booléen	39	l'argument est-il inconnu ?
`urcorner`	–	dessin chemin stylo	paire	50	coin supérieur droit de la *bounding box*
†`whatever`	–	–	numérique	30	crée une nouvelle inconnue anonyme
`xpart`	–	paire transformation	nombre	40	composante x ou t_x d'une transformation

TAB. 9: Opérateurs (quatrième partie) (suite)

Nom	Types arguments/sortie			Page	Explication
	Gauche	**Droite**	**Résultat**		
`xscaled`	dessin chemin paire stylo trans-forma-tion	numéri-que	dessin chemin paire stylo trans-forma-tion	60	multiplie l'abscisse par la quantité donnée
`xxpart`	–	trans-forma-tion	nombre	63	composante t_{xx} d'une transformation
`xypart`	–	trans-forma-tion	nombre	63	composante t_{xy} d'une transformation
`ypart`	–	trans-forma-tion	nombre		composante y ou t_y d'une transformation
`yscaled`	dessin chemin paire stylo trans-forma-tion	numéri-que	dessin chemin paire stylo trans-forma-tion	60	multiplie l'ordonnée par la quantité donnée
`yxpart`	–	trans-forma-tion	nombre	40	composante t_{yx} d'une transformation
`yypart`	–	trans-forma-tion	nombre	63	composante t_{yy} d'une transformation
`zscaled`	dessin chemin paire stylo trans-forma-tion	paire	dessin chemin paire stylo trans-forma-tion	60	effectue une rotation et une homothétie sur toutes les coordon-nées de telle sorte que (1,0) soit transformé en la paire donnée ; cela revient à effectuer une multiplication complexe

TAB. 10: Commandes

Nom	**Page**	**Explication**
`addto`	73	commande de bas niveau pour les tracés et les remplissages
`clip`	74	applique un chemin de détourage à un dessin
†`cutdraw`	90	trace avec des extrémités carrées
†`draw`	20	trace une ligne ou un dessin
†`draw-arrow`	69	trace une ligne avec une flèche à la fin
†`draw-dblarrow`	69	trace une ligne avec des flèches aux deux extrémités
†`fill`	51	remplit l'intérieur d'un chemin cyclique
†`filldraw`	70	trace un chemin cyclique et remplit l'intérieur
`interim`	78	rend locale une modification d'une variable interne
`let`	--	assigne à un *token* la signification d'un autre
†`loggin-all`	104	active toutes les informations
`newinter-nal`	43	déclare une nouvelle variable interne
†`pickup`	35	spécifie une nouvelle plume pour les tracés
`save`	77	rend une variable locale
`setbounds`	50	fait tenir un dessin dans sa boîte frontière
`shipout`	75	commande de bas niveau pour envoyer une figure
`show`	33	affiche les expressions sous forme symbolique
`showde-penden-cies`	104	affiche toutes les équations non résolues
`showtoken`	103	affiche une explication sur le rôle d'un *token*
`showva-riable`	103	affiche les variables symboliquement
`special`	131	écrit une chaîne directement dans le fichier POSTSCRIPT de sortie
†`tracing-all`	104	active toutes les informations
†`tracing-none`	104	désactive toutes les informations
†`undraw`	70	efface une ligne ou un dessin
†`unfill`	53	efface l'intérieur d'un chemin cyclique
†`unfill-draw`	70	efface un chemin cyclique et son intérieur

TAB. 11: Macros-fonctions

Nom	Arguments	Résultat	Page	Explication
‡boxit	suffixe, dessin	–	95	définit une boîte contenant le dessin
‡boxit	suffixe, chaîne	–	99	définit une boîte contenant le texte
‡boxit	suffixe, <vide>	–	99	définit une boîte vide
‡boxjoin	équations	–	96	donne les équations de connexion de boîtes
‡bpath	suffixe	chemin	95	frontière d'une boîte
†build-cycle	liste de chemins	chemin	53	construit un chemin cyclique
‡circleit	suffixe, dessin	–	99	place un dessin dans une boîte circulaire
‡circleit	suffixe, chaîne	–	99	place un texte dans une boîte circulaire
‡circleit	suffixe, <vide>	–	99	définit une boîte circulaire vide
†dash-pattern	distances on/off	dessin	65	crée un motif pour les lignes discontinues
†decr	variable numérique	numérique	90	décrémente et retourne la nouvelle valeur
†dotlabel	suffixe, dessin, paire	–	44	affiche un point et trace un dessin à côté
†dotlabel	suffixe, chaîne, paire	–	44	affiche un point et affiche un texte à côté
†dotlabels	suffixe, numéros de points	–	45	affiche un point sur une coordonnée z et affiche leur numéro
‡drawboxed	liste de suffixes	–	95	trace les boîtes nommées et leurs contenus
‡drawboxes	liste de suffixes	–	97	trace les boîtes nommées
†drawoptions	options de tracé	–	70	active les options de commandes de tracé
‡drawunboxed	liste de suffixes	–	97	trace le contenu des boîtes nommées
‡fixpos	liste de suffixes	–	97	trouve les tailles et les positions des boîtes nommées

TAB. 11: Macros-fonctions (suite)

Nom	Arguments	Résultat	Page	Explication
‡fixsize	liste de suffixes	–	97	trouve les tailles des boîtes nommées
†incr	variable numérique	numérique	90	incrémente et retourne la nouvelle valeur
†label	suffixe, dessin, paire	–	44	trace un dessin proche du point donné
†label	suffixe, chaîne, paire	–	44	affiche un texte proche du point donné
†labels	suffixe, numéros de points	–	45	affiche les numéros des points z sans point affiché
†max	liste de numériques	numérique	–	trouve le maximum
†max	liste de chaînes	chaîne	–	trouve la dernière chaîne dans l'ordre lexicographique
†min	liste de numériques	numérique	–	trouve le minimum
†min	liste de chaînes	chaîne	–	trouve la première chaîne dans l'ordre lexicographique
‡pic	suffixe	dessin	95	contenu d'une boîte décalée vers la position donnée
†thelabel	suffixe, dessin, paire	dessin	45	dessin décalé comme s'il était la légende d'un point
†thelabel	suffixe, chaîne, paire	dessin	45	texte positionné comme s'il était la légende d'un point
†z	suffixe	paire	40	la paire (x<suffixe>,y<suffixe>)

<atome>→ <variable> | <argument>
| <nombre ou fraction>
| <variable interne>
|`(`<expression>`)`
|`begingroup`<liste d'instructions><expression>`endgroup`
| <opérateur isolé>
| <pseudo fonction>
<primaire>→ <atome>
|`(`<expression numérique>`,`<expression numérique>`)`
|`(`<expression numérique>`,`<expression numérique>`,`<expression numérique>`)`
| <opérateur of><expressions>`of`<primaire>
| <opérateur unaire><primaire>
|`str`<suffixe>
|`z`<suffixe>
| <atome numérique>`[`<expression>`,`<expression>`]`
| <opérateur de multiplication scalaire><primaire>
<secondaire> → <primaire>
| <secondaire><opérateur binaire><primaire>
| <secondaire><transformation>
<tertiaire>→ <secondaire>
| <tertiaire><opérateur binaire secondaire><secondaire>
<sous-expression>→ <tertiaire>
| <expression chemin><connecteur><nœud de chemin>
<expression>→ <sous-expression>
| <expression><opérateur binaire tertiaire><tertiaire>
| <sous-expression chemin><spécificateur de direction>
| <sous-expression chemin><connecteur>`cycle`

<nœud de chemin>→ <tertiaire>
<connecteur> → `- -`
| <spécificateur de direction><connecteur élémentaire><spécificateur de direction>
<spécificateur de direction>→ <vide>
|`{curl`<expression numérique>`}`
|`{`<expression numérique>`,`<expression numérique>`}`
<connecteur élémentaire>→ `..`
|`...`
|`..`<tension>`..`
|`..`<contrôles>`..`
<tension>→ `tension`<numérique primaire>
|`tension`<numérique primaire>`and`<numérique primaire>
<contrôles>→ `controls`<paire primaire>
|`controls`<paire primaire>`and`<paire primaire>

<argument>→ <*token* symbolique>
<nombre ou fraction>→ <nombre>`/`<nombre>
| <nombre non suivi de '`/`<nombre>'>
<opérateur de multiplication scalaire>→ +|−
| <'<nombre ou fraction>' non suivi de '<opérateur additif><nombre>'>

FIG. 55 – Syntaxe des expressions (première partie)

<transformation>→`rotated`<numérique primaire>
|`scaled`<numérique primaire>
|`shifted`<paire primaire>
|`slanted`<numérique primaire>
|`transformed`<transformation primaire>
|`xscaled`<numérique primaire>
|`yscaled`<numérique primaire>
|`zscaled`<paire primaire>
|`reflectabout(`<expression paire>`,`<expression paire>`)`
|`rotatedaround(`<expression paire>`,`<expression numérique>`)`

<opérateur isolé>→ `false`|`normaldeviate`|`nullpicture`|`pencircle`
|`true`|`whatever`
<opérateur unaire>→ <type>
|`abs`|`angle`|`arclength`|`ASCII`|`bbox`|`bluepart`|`bot`|`ceiling`
|`center`|`char`|`cosd`|`cycle`|`decimal`|`dir`|`floor`|`fontsize`
|`greenpart`|`hex`|`inverse`|`know`|`length`|`lft`|`llcorner`
|`lrcorner`|`makepath`|`makepen`|`mexp`|`mlog`|`not`|`oct`|`odd`
|`redpart`|`reverse`|`round`|`rt`|`sind`|`sqrt`|`top`|`ulcorner`
|`uniformdeviate`|`unitvector`|`unknow`|`urcorner`|`xpart`|`xxpart`
|`xypart`|`ypart`|`yxpart`|`yypart`
<type>→ `boolean`|`color`|`numeric`|`pair`
|`path`|`pen`|`picture`|`string`|`transform`
<opérateur binaire primaire>→ `*`|`/`|`**`|`and`|`dotprod`|`div`|`infont`|`mod`
<opérateur binaire secondaire>→ `+`|`-`|`++`|`+-+`|`or`|
`intersectionpoint`|`intersectiontimes`
<opérateur binaire tertiaire>→ |`&`|`<`|`<=`|`<>`|`=`|`>`|`>=`
|`cutafter`|`cutbefore`
<opérateur of>→`arctime`|`direction`|`directionpoint`|`directiontime`
|`penoffset`|`point`|`postcontrol`|`precontrol`|`subpath`
|`substring`

<variable>→ <*tag*><suffixe>
<suffixe>→ <vide> | <suffixe><indice> | <suffixe><*tag*>
| <paramètre suffixe>
<indice>→ <nombre> |`[`<expression numérique>`]`

<variable interne>→ `ahangle`|`ahlength`|`bboxmargin`
|`charcode`|`day`|`defaultpen`|`defaultscale`|`labeloffset`
|`linecap`|`linejoin`|`miterlimit`|`month`|`pausing`
|`prologues`|`showstopping`|`time`|`tracingoutput`
|`tracingcapsules`|`tracingchoices`|`tracingcommands`
|`tracingequations`|`tracinglostchars`|`tracingmacros`
|`tracingonline`|`tracingrestores`|`tracingspecs`
|`tracingstats`|`tracingtitles`|`truecorners`
|`warningcheck`|`year`
| <*token* symbolique défini par `newinternal`>

FIG. 56 – Syntaxe des expressions (deuxième partie)

<pseudo-fonction>→ `min(`<liste d'expressions>`)`
|`max(`<liste d'expressions>`)`
|`incr(`<variable numérique>`)`
|`decr(`<variable numérique>`)`
|`dashpattern(`<liste on/off>`)`
|`interpath(`<expression numérique>`,`<expression numérique>`,`
<expression numérique>`)`
|`buildcycle(`<liste d'expressions chemin>`)`
|`thelabel`<suffixe>`(`<expression>`,`<expression paire>`)`
<liste d'expressions chemin>→ <expression chemin>
| <liste expressions chemin><expression chemin>
<liste on/off>→ <liste on/off><clause on/off> | <clause on/off>
<clause on/off>→ `on`<numérique tertiaire> |`off`<numérique tertiaire>

FIG. 57 – Syntaxe des macros-fonctions

<atome numérique>→ <expression>
<expression booléenne>→ <expression>
<expression chaîne>→ <expression>
<expression chemin>→ <expression>
<expression couleur>→ <expression>
<expression dessin>→ <expression>
<expression numérique>→ <expression>
<expression paire>→ <expression>
<expression stylo>→ <expression>
<paramètre suffixe>→ <paramètre>
<primaire numérique>→ <primaire>
<paire primaire>→ <primaire>
<transformation primaire>→ <primaire>
<sous-expression chemin>→ <sous-expression>
<numérique tertiaire>→ <tertiaire>
<variable dessin>→ <variable>
<variable numérique>→ <variable> | <variable interne>

FIG. 58 – Productions diverses pour compléter la grammaire BNF

<programme>→ <liste d'instructions>`end`
<liste d'instructions>→ <vide> | <liste d'instructions> `;`<instruction>
<instruction>→ <vide>
| <équation> | <assignation>
| <déclaration> | <définition de macro>
| <instruction composée> | <pseudo-procédure>
| <commande>
<instruction composée>→ `begingroup`<liste d'instructions>`endgroup`
|`beginfig(`<expression numérique>`);`<liste d'instructions>`;endfig`

<équation>→ <expression>`=`<côté droit>
<assignation>→ <variable> `:=`<côté droit>
| <variable interne> `:=`<côté droit>
<côté droit>→ <expression> | <équation> | <assignation>

<déclaration>→ <type><liste de déclaration>
<liste de déclaration>→ <variable générique>
| <liste de déclaration>`,`<variable générique>
<variable générique>→ <*token* symbolique><suffixe générique>
<suffixe générique>→ <vide> | <suffixe générique><*tag*>
| <suffixe générique>`[]`

<définition de macro>→ <en-tête de macro>`=`<texte de remplacement>`enddef`
<en-tête de macro>→ `def`<*token* symbolique><partie délimitée><partie illimitée>
|`vardef`<variable générique><partie délimitée><partie illimitée>
|`vardef`<variable générique>`@#`<partie délimitée><partie illimitée>
| <def binaire><paramètre><*token* symbolique><paramètre>
<partie délimitée>→ <vide>
| <partie délimitée>`(`<type de paramètre><paramètres *tokens*>`)`
<type de paramètre>→ `expr`|`suffix`|`text`
<paramètres *tokens*>→ <*token* symboliques>
<partie illimitée>→ <vide>
| <type de paramètre><paramètre>
| <niveau de priorité><paramètre>
|`expr`<paramètre>`of`<paramètre>
<niveau de priorité>→ `primary`|`secondary`|`tertiary`
<def binaire>→ `primarydef`|`secondarydef`|`tertiarydef`

<pseudo-procédure>→ `drawoptions(`<liste d'options>`)`
|`label`<suffixe d'étiquette> |`(`<expression>`,`<expression paire>`)`
|`dotlabel`<suffixe d'étiquette>`(`<expression>`,`<expression paire>`)`
|`labels`<suffixe d'étiquette>`(`<liste de numéros de points>`)`
|`dotlabels`<suffixe d'étiquette>`(`<liste de numéros de points>`)`
<liste de numéros de points>→ <suffixe> | <liste de numéros de points>`,`<suffixe>
<suffixe de légende>→ <vide> |`lft`|`rt`|`top`|`bot`|`ulft`|`urt`|`llft`|`lrt`

FIG. 59 – Syntaxe générale pour les programmes METAPOST

<commande> → `clip`<variable dessin>`to`<expression chemin>
|`interim`<variable interne> `:=`<côté droit>
|`let`<*token* symbolique> `:=`<côté droit>
|`newinternal`<liste de *tokens* symboliques>
|`pickup`<expression>
|`randomseed :=`<expression numérique>
|`save`<liste de *tokens* symboliques>
|`setbouds`<variable dessin>`to`<expression chemin>
|`shipout`<expression dessin>
|`special`<expression chaîne>
| <commande addto>
| <commande de tracé>
| <commande de métrique de fonte>
| <commande show>
| <commande d'information>

<commande show>→`show`<liste d'expressions>
|`showvariable`<liste de *tokens* symboliques>
|`showtoken`<liste de *tokens* symboliques>
|`showdependencies`

<liste de *tokens* symboliques>→ <*token* symbolique>
| <*token* symbolique>`,`<liste de *tokens* symboliques>
<liste d'expressions>→ <expression> | <liste d'expressions>`,`<expression>

<commande addto>→ `addto`<variable dessin>`also`<expression dessin>
<liste d'options>
|`addto`<variable dessin>`contour`<expression chemin><liste d'options>
|`addto`<variable dessin>`doublepath`<expression chemin>
<liste d'options>
<liste d'options>→ <vide> | <option de tracé><liste d'options>
<option de tracé>→ `withcolor`<expression couleur>
|`withpen`<expression stylo> |`dashed`<expression dessin>

<commande de tracé>→ `draw`<expression dessin><liste d'option>
| <type de remplissage><expression chemin><liste d'options>
<type de remplissage>→ `fill`|`draw`|`filldraw` |`unfill`|`undraw`|`unfilldraw`
|`drawarrow`|`drawdblarrow`|`cutdraw`

<commande d'informations>→ `tracingall`|`loggingall`|`tracingnone`

FIG. 60 – Syntaxe des commandes

<test if>→ `if`<expression booléenne> `:`<*tokens* équilibrés><alternatives>`fi`
<alternatives>→ <vide>
|`else :`<*tokens* équilibrés>
|`elseif`<expression booléenne> `:` <*tokens* équilibrés><alternatives>

<boucle>→ <en-tête de boucle> `:`<texte de boucle>`endfor`
<en-tête de boucle>→ `for`<*token*>`=`<progression>
|`for`<*token*>`=`<liste for>
|`forsuffixes`<*token*>`=`<liste de suffixes>
|`forever`
<progression>→ <expression numérique>`upto`<expression numérique>
| <expression numérique>`upto`<expression numérique>
| <expression numérique>`downto`<expression numérique>
<liste for>→ <expression> | <liste for>`,`<expression>
<liste de suffixes>→ <suffixe> | <liste de suffixes>`,`<suffixe>

FIG. 61 – Syntaxe des tests et des boucles

B. Comparaison METAPOST – METAFONT

Les langages METAPOST et METAFONT ont tellement de points communs que les utilisateurs confirmés de METAFONT voudront sans doute laisser de côté la plupart des explications de ce document et se concentrer sur les concepts qui sont uniques à METAPOST. Les comparaisons de cet appendice sont prévues pour aider les experts familiers du *METAFONTbook* ainsi que les autres utilisateurs qui tireront profit des explications plus détaillées de KNUTH [4].

Puisque METAFONT est prévu pour construire les fontes de TEX, il possède un certain nombre de primitives permettant de produire des fichiers `tfm` dont TEX a besoin pour connaître les dimensions des caractères, les informations d'espacement, les ligatures et le crénage. METAPOST peut également être utilisé pour créer des fontes et il possède également les primitives de METAFONT pour construire les fichiers `tfm`. Elles sont résumées dans la table 12. On peut trouver des explications dans la documentation de METAFONT [4, 7].

TAB. 12 – Primitives METAPOST pour construire des fichiers `tfm`

commandes	`charlist`, `extensible`, `fontdimen`, `headerbyte`, `kern`, `ligtable`
opérateurs de ligature	::, `= :`, =: |, =: | >, =:, | =:>, | =: |, | := | >, | =: | > >, || :
autres opérateurs	`charexists`

Bien que METAPOST possède les primitives permettant de produire des fontes, beaucoup de primitives et de variables internes pour la génération de fontes définies dans Plain METAFONT ne font pas partie de Plain METAPOST. Pour cela, il y a un *package* séparé de macros, appelé `mfplain`, qui définit les macros requises pour permettre à METAPOST de compiler les fontes *Computer Modern* de KNUTH comme le montre la table 13 [6][10]. Pour charger ces macros, il suffit d'écrire `&mfplain` avant le nom du fichier source. Cela peut être effectué également au niveau de l'invite `**` après avoir lancé l'interpréteur METAPOST sans argument, ou à partir d'une ligne de commande qui pourrait ressembler à

```
mp '&mfplain' cmr10
```

L'analogue de la commande METAFONT suivante

```
mf '\ mode=lowres; mag=1.2; input cmr10'
```

10 La syntaxe des lignes de commandes dépend du système. Les apostrophes sont nécessaires sur la plupart des systèmes Unix pour protéger des caractères spéciaux tels que &.

est

```
mp '&mfplain \ mode=lowres; mag=1.2; input cmr10'
```

Le résultat est un ensemble de fichiers POSTSCRIPT, un pour chaque caractère de la fonte. Un certain travail sera nécessaire pour pouvoir les incorporer dans des fontes POSTSCRIPT de type 3 [1].

TAB. 13: Macros et variables internes définies uniquement dans le package `mfplain`

Définies dans le package `mfplain`	
`beginchar`	`font_identifier`
`blacker`	`font_normal_shrink`
`capsule_def`	`font_normal_space`
`change_width`	`font_normal_stretch`
`define_blacker_pixels`	`font_quad`
`define_corrected_pixels`	`font_size`
`define_good_x_pixels`	`font_slant`
`define_good_y_pixels`	`font_x_height`
`define_horizontal_corrected_pixels`	`italcorr`
`define_pixels`	`labelfont`
`define_whole_blacker_pixels`	`makebox`
`define_whole_pixels`	`makegrid`
`define_whole_vertical_blacker_pixels`	`maketicks`
`define_whole_vertical_pixels`	`mode_def`
`endchar`	`mode_setup`
`extra_beginchar`	`o_correction`
`extra_endchar`	`proofrule`
`extra_setup`	`proofrulethickness`
`font_coding_scheme`	`rulepen`
`font_extra_space`	`smode`
Définies comme non-op dans le package `mfplain`	
`cullit`	`proofoffset`
`currenttransform`	`screenchars`
`gfcorners`	`screenrule`
`grayfont`	`screenstrokes`
`hround`	`showit`
`imagerules`	`slantfont`
`lowres_fix`	`titlefont`
`nodisplays`	`unitpixel`
`notransforms`	`vround`
`openit`	

Une autre limitation du package `mfplain` est que certaines variables internes de Plain METAFONT ne peuvent pas être traduites de façon sensée en METAPOST. Ceci inclue `displaying`, `currentwindow`, `screen_rows` et `screen_cols` qui dépendent de la possibilité qu'a METAFONT d'afficher des images sur un écran d'ordinateur. De plus, `pixels_per_inch` est irrecevable puisque METAPOST fonctionne avec des unités fixées à partir des points POSTSCRIPT.

La raison pour laquelle certaines macros et variables internes n'ont aucun sens avec METAPOST est que les commandes primitives de METAFONT `cull`, `display`, `openwindow`, `numspecial` et `totalweight` ne sont pas implémentées dans METAPOST. Ne sont également pas implémentées un certain nombre de variables internes ainsi que l'<option de tracé> `withweight`. Voici une liste complète des variables internes dont la signification première dans METAFONT ne peut pas avoir de sens dans METAPOST :

```
autorounding  fillin   proofing     tracingpens
xoffset       chardx   granularity  smoothing
turningcheck  yoffset  chardy       hppp
tracingeges   vppp
```

Il y a également une primitive de METAFONT qui a un sens légèrement différent dans METAPOST. Les deux langages permettent des instructions de la forme

`special`<expression chaîne>`;`

mais METAFONT copie la chaîne dans un fichier de sortie « fonte générique » tandis que METAPOST interprète la chaîne comme une séquence de commandes POSTSCRIPT qui seront placées au début du prochain fichier de sortie.

Toutes les autres différences entre METAFONT et METAPOST proviennent de fonctionnalités trouvées uniquement dans METAPOST. Elles sont listées à la table 14. Les seules commandes présentées ici qui n'ont pas été étudiées dans les chapitres précédents sont `extra_beginfig`, `extra_endfig` et `mpxbreak`. Les deux premières sont des chaînes qui contiennent des commandes supplémentaires devant être exécutées par `beginfig` et `endfig` exactement comme `extra_beginchar` et `extra_endchar` étaient exécutées par `beginchar` et `endchar` (le fichier `boxes.mp` utilise cette possibilité).

L'autre fonctionnalité listée dans la table 14 non répertoriée dans l'index est `mpxbreak`. Elle est utilisée pour séparer des blocs traduits en commandes TEX ou troff dans des fichiers `mpx`. Cette commande ne devrait pas concerner un utilisateur humain puisque les fichiers `mpx` sont généralement construits de façon automatique.

TAB. 14: Macros et variables internes définies par METAPOST et absentes de METAFONT

Primitives METAPOST **non trouvées dans** METAFONT		
`bluepart`	`infont`	`redpart`
`btex`	`linecap`	`setbounds`
`clip`	`linejoin`	`tracinglostchars`
`color`	`llcorner`	`truecorners`
`dashed`	`lrcorner`	`ulcorner`
`etex`	`miterlimit`	`urcorner`
`fontsize`	`mpxbreak`	`verbatimtex`
`greenpart`	`prologues`	`withcolor`
Variables et macros définies uniquement dans METAPOST		
`ahangle`	`cutbefore`	`extra_beginfig`
`ahlength`	`cuttings`	`extra_endfig`
`background`	`dashpattern`	`green`
`bbox`	`defaultfont`	`label`
`bboxmargin`	`defaultpen`	`labeloffset`
`beginfig`	`defaultscale`	`mitered`
`beveled`	`dotlabel`	`red`
`black`	`dotlabels`	`rounded`
`blue`	`drawarrow`	`squared`
`buildcycle`	`drawdblarrow`	`thelabel`
`butt`	`drawoptions`	`white`
`center`	`endfig`	
`cutafter`	`evenly`	

Bibliographie

[1] Adobe Systems Inc. POSTSCRIPT *Language Reference Manual*, Addison Wesley, Reading, Massachusets, 1986.

[2] J. D. HOBBY, Smooth, easy to compute interpolating splines, *Discrete and Computational Geometry*, 1(2), 1986.

[3] Brian W. KERNIGHAN, Pic – a graphics language for typesetting, In *Unix Research System Papers, Tenth Edition*, pages 53–77, AT&T Bell Laboratories, 1990.

[4] D. E. KNUTH, *The METAFONTbook*, Addison Wesley, Reading, Massachusets, 1986. Volume C of *Computers and typesetting*.

[5] D. E. KNUTH, *The TEXbook*, Addison Wesley, Reading, Massachusets, 1986. Volume C of *Computers and Typesetting*.

[6] D. E. KNUTH, *Computer Modern Typefaces*, Addison Wesley, Reading, Massachusets, 1986. Volume A of *Computers and Typesetting*.

[7] D. E. KNUTH, The new versions of TEX and METAFONT, *TUGboat, the TEX User's Group Newsletter*, 10(3) : 325 – 328, November 1989.

Index