Une introduction à MetaPost

http://pauillac.inria.fr/~cheno/metapost/

Laurent Chéno
laurent.cheno@inria.fr

Toulouse — mai 1999

Nous présentons ici une introduction à MetaPost, l'outil créé par John Hobby à partir de Metafont, qui utilise un langage de description d'images et produit des fichiers PostScript.
Après une description non exhaustive certes, mais qui tend à l'essentiel, nous présentons quelques applications sous la forme d'exemples qui démontrent la flexibilité et l'efficacité de MetaPost.

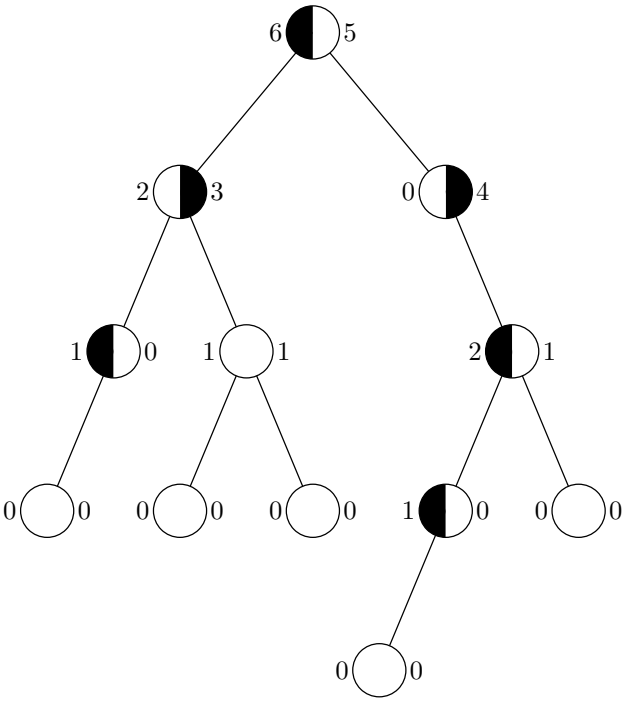


Table des matires

1 Description de MetaPost

1.1 Fonctionnement

Un fichier source pour MetaPost, disons `foo.mp`, a la structure suivante :

```
prologues := 2 ;
input boxes ;
verbatimtex \input mes_definitions_TeX.tex etex

beginfig(1)
   ...
   ...
endfig ;

beginfig(2)
   ...
   ...
endfig ;

end
```

Un tel fichier contient la description de deux figures PostScript que produira MetaPost avec les noms `'foo.1'` et `'foo.2'`. On aura noté la possibilité d'insérer un fichier (ici `boxes`, d'ailleurs fourni avec la distribution standard de MetaPost) de définitions MetaPost, et même de poser des définitions TEX valides pour tout le fichier (c'est le rôle du `verbatimtex`).
Il suffit alors, dans un fichier LATEX, d'insérer une commande d'insertion de graphique EPS pour chacun des fichiers PostScript obtenus.

1.2 Un survol des types

MetaPost connaît neuf types :

numeric est le type des nombres : ils sont représentés par des multiples entiers de 1/65536, et leur valeur absolue ne peut dépasser 4096 (même si elle peut aller jusqu'à 32768 dans les calculs intermédiaires). S'ils représentent des longueurs, elles sont comptées par défaut en point PostScript (le `bp` de TEX et de MetaPost). Les autres unités usuelles sont définies par les équations suivantes : `1in=2.54cm=72.27pt=72bp=25.4mm`. Notons que, par exemple, la variable `in` est égale à 72, tout simplement, et que la notation `3in` utilise une multiplication implicite. Pour terminer, remarquons que seules les variables de type `numeric` n'ont pas besoin d'être déclarées, même si une déclaration `numeric l,m,n ;` est tout à fait valide.

pair est le type des couples de nombres, et est d'un usage constant pour les coordonnées d'un point. On écrira par exemple `z0=(0,3cm) ; z1=(2pt,-4in) ;`

path est le type des chemins : si `p` désigne un chemin, on pourra le tracer (avec la plume courante) par l'instruction `draw p`, on pourra dans le cas d'un chemin fermé colorier en noir (ou une autre couleur) le domaine qu'il délimite par l'instruction `fill p..cycle` (j'ai ajouté le mot-clé `cycle` pour être sûr de bien refermer le chemin, et éviter que `fill` ne déclenche une erreur).

transform est le type des transformations géométriques : il permet de représenter toutes les transformations affines, et peut s'appliquer indifféremment à un objet de type `pair`, `path`, `picture` ou même `pen`.

color est le type des couleurs : elles sont représentées par un triplet des composantes RVB. `black` est $(0, 0, 0)$, alors que `white` est $(1, 1, 1)$. Un gris comme $(0.4, 0.4, 0.4)$ peut être désigné par `0.4white`. Sont également prédéfinies les couleurs `red`, `green` et `blue`, mais on pourra écrire par exemple `yellow=red+green`.

string est le type des chaînes de caractères.

boolean est le type des booléens. Sont prédéfinies les deux constantes `true` et `false`, et les opérateurs habitules `not`, `and` et `or`.

picture est le type des images : une instruction comme `draw` ajoute en fait à la variable globale `currentpicture` le nouveau tracé. Une image peut être transformée, ou ajoutée à une autre.

pen est le type des plumes : la plume par défaut, `currentpen`, est circulaire et est définie par `pencircle scaled 0.5bp`. L'opérateur `pickup`, appliqué à un objet du type `pen`, permet de sélectionner une nouvelle plume, comme par exemple dans l'instruction `pickup pencircle scaled 4bp` qui permet de tracer par la suite avec un trait épais.

1.3 Opérateurs

1.3.1 Opérateurs pour le type `numeric`

Outre les opérations arithmétiques habituelles, MetaPost offre l'opérateur `**` d'exponentiation, la racine carrée `sqrt`, ou encore `abs`, `round`, `floor`, `ceiling`, et, pour la trigonométrie, `sind` et `cosd` qui attendent des arguments en degrés. De plus, on dispose des opérations pythagoriciennes a `++` b représente $\sqrt{a^2+b^2}$ et a `+-+` b représente $\sqrt{a^2-b^2}$.
La plus grosse difficulté provient des règles très inhabituelles de priorité qu'utilise MetaPost à l'instar de ce qu'a choisi Knuth pour Metafont.
Ainsi, par exemple, `3*a**2` désigne $(3a)^2$. De même `sqrt 2/3` désigne $\sqrt{2/3}$ alors que `sqrt(1+1)/3` désigne $\sqrt{2}/3$.

1.3.2 Opérateurs pour le type `pair`

On peut à l'envie ajouter ou soustraire des couples de coordonnées, comme on peut les multiplier par un scalaire.
On dispose de la notation $2/3[a,b]$ pour désigner $2/3a + 1/3b$, c'est-à-dire un barycentre de a et b. On a plus généralement $t[a,b] = ta + (1-t)b = a + (1-t)(b-a)$.
Notons que cette notation est applicable également aux couleurs, ce qui permet par exemple de réaliser de jolis dégradés.
L'opérateur `abs` renvoie la norme euclidienne d'un vecteur, et `unitvector` renvoie le vecteur argument divisé par sa norme, de sorte que l'égalité `abs(u)*unitvector(u)=u` est toujours vraie. L'opérateur `angle` renvoie l'angle polaire d'un vecteur, c'est l'inverse de l'opérateur `dir` : il existe des abréviations commodes pour `right=dir 0`, `up=dir 90`, `left=dir 180` ou `down=dir 270`.

1.3.3 Autres opérateurs

L'opérateur `&` est utilisé aussi bien pour la concaténation des chaînes de caractères que pour la concaténation des chemins (qu'on suppose effectivement adjacents : l'extrémité du premier est égale à l'origine du second).
Citons également l'opérateur de sélection d'une sous-chaîne : `substring (2,4) of "abcdef"` renvoie la sous-chaîne `"cd"` (caractères d'indices 2 (inclus) à 4 (exclus) de la chaîne complète, la numérotation commençant à 0).
Chacun des noms de type est également un opérateur à un argument et à résultat booléen, qui spécifie si l'argument est du type indiqué.
Enfin, il existe des sélecteurs : `xpart` et `ypart` renvoient les coordonnées d'un couple, et `redpart`, `greenpart` et `bluepart` renvoient les coordonnées colorimétriques d'un triplet de type `color`.
Pour terminer, mais nous en reparlerons quand nous verrons les noms tout à l'heure, il faut savoir que par défaut MetaPost connaît l'abréviation `z`.*suffixe* pour un couple (`x`.*suffixe*, `y`.*suffixe*) : ainsi a-t-on les égalités `z0=(x0,y0)`, `z24=(x24,y24)`, ou même `z.a=(x.a,y.a)`.

1.4 Définition des chemins

Les chemins de MetaPost peuvent être des lignes brisées ou des courbes de Bézier, ou un mélange des deux.

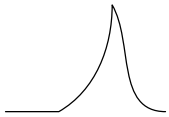
1.4.1 Lignes brisées

On définira une ligne brisée en séparant les sommets successifs par `--`, et on pourra la refermer en un chemin fermé en concluant par `--cycle`.
Par exemple :

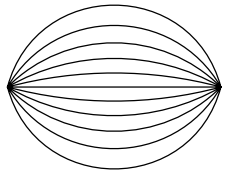
```
beginfig(1) ;
  draw (0,0)--(10,10)--(20,0)--(10,-10)--cycle ;
  draw (0,-10)--(20,-10)--(20,10)--(0,10)--cycle ;
endfig ;
```

1.4.2 Courbes de Bézier

Si au lieu de `--` on utilise l'opérateur `..`, on obtient une courbe de Bézier qui passe par les points indiqués. Heureusement, on peut paramétrer bien davantage les courbes obtenues, en précisant les tangentes, voire les demi-tangentes, aux points concernés, comme ci-dessous.

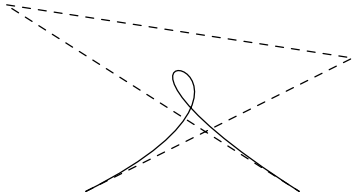


```
beginfig(2) ;
    draw (0,0)--(20,0){dir30}..{up}(40,40){dir-60}..{right}(60,0) ;
    for i=-5 upto 5:
        draw (0,-80){dir 15i}..(80,-80) ;
    endfig ;
endfig ;
```



D'autres réglages sont disponibles, dont nous ne parlerons pas ici. Terminons simplement en signalant la possibilité de définir complètement la courbe de Bézier désirée, en spécifiant les points de contrôle :

```
beginfig(3) ;
    draw (0,0).. controls (100,50) and (-30,70) .. (80,0) ;
    draw (0,0)--(100,50)--(-30,70)--(80,0) dashed evenly ;
endfig ;
```



1.5 Transformations

MetaPost permet, on l'a déjà dit, de manipuler des transformations affines du plan.
Le tableau suivant fait un résumé des transformations élémentaires qui sont autorisées ; rappelons qu'on peut les appliquer à un couple de coordonnées, à un chemin, une image ou une plume.

(x,y) `shifted` (a,b)	$=$	$(x+a, y+b)$
(x,y) `rotated` θ	$=$	$(x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta)$
(x,y) `slanted` a	$=$	$(x+ay, y)$
(x,y) `scaled` a	$=$	(ax, ay)
(x,y) `xscaled` a	$=$	(ax, y)
(x,y) `yscaled` a	$=$	(x, ay)
(x,y) `zscaled` (a,b)	$=$	$(ax-by, bx+ay)$

Ces opérateurs peuvent bien entendu être combinés.
La transformation identique est nommée `identity`.
On peut obtenir la transformation réciproque d'une transformation `T` ou bien en utilisant l'opérateur `inverse` ou bien en posant l'équation suivante : `identity = invT transformed T`.
Citons pour mémoire, bien que cela soit d'un usage peu pratique, la possibilité de sélectionner les six coefficients caractéristiques d'une transformation `T` en explicitant l'égalité matricielle suivante :

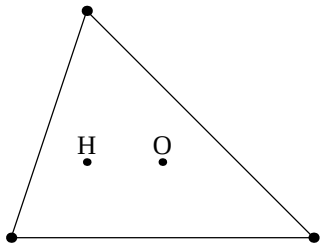
$$\begin{pmatrix} x \\ y \end{pmatrix} \texttt{transformed T} = \begin{pmatrix} \texttt{xxpart T} & \texttt{xypart T} \\ \texttt{yxpart T} & \texttt{yypart T} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \texttt{xpart T} \\ \texttt{ypart T} \end{pmatrix}$$

C'est ainsi qu'on pourra par exemple spécifier que `T` est une similitude directe en écrivant les seules équations suivantes : `xxpart T = yypart T; xypart T = -yxpart T`.

1.6 Équations

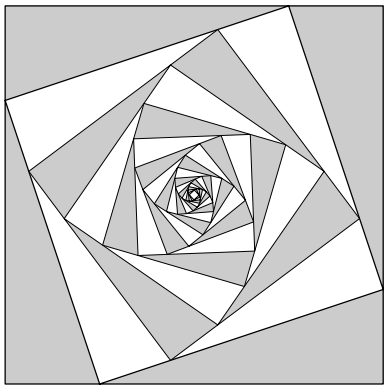
Un des immenses intérêts de MetaPost est sa facilité à gérer des systèmes d'équations linéaires. On en verra différentes applications dans les exemples ci-dessous.
Mais donnons ici un premier exemple, qui donne un aperçu de ces possibilités, en construisant quelques points remarquables du triangle.
Le mot-clé `whatever` désigne un nombre inconnu : écrire `a=whatever[b,c]` c'est donc dire que `a` est sur la droite `(bc)`.

```
beginfig(4) ;
    % tracé du triangle
    z0 = (0,0) ; z1 = (4cm,0) ; z2 = (1cm,3cm) ;
    draw z0--z1--z2--cycle ;
    for i=0 upto 2:
        draw z[i] withpen pencircle scaled 4bp ;
    endfor

    % recherche de l'orthocentre H
    (z3 - z0) rotated 90 = whatever*(z2 - z1) ;
    (z3 - z1) rotated 90 = whatever*(z0 - z2) ;
    dotlabel.top("H",z3) ;

    % recherche du centre O du cercle circonscrit
    (z4 - 1/2[z0,z1]) rotated 90 shifted z0 = whatever[z0,z1] ;
    (z4 - 1/2[z1,z2]) rotated 90 shifted z1 = whatever[z1,z2] ;
    dotlabel.top("O",z4) ;
endfig ;
```



On peut s'amuser aussi à laisser MetaPost résoudre un système dont l'inconnue est une transformation :

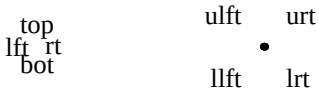
```
beginfig(5) ;
    z0 = (0,0) ; z1 = (5cm,0) ; z2 = (5cm,5cm) ; z3 = (0,5cm) ;
    transform T ;
    z0 transformed T = 1/4[z0,z1] ;
    z1 transformed T = 1/4[z1,z2] ;
    z2 transformed T =  1/4[z2,z3] ;

    path carre ;
    carre = z0--z1--z2--z3--cycle ;

    fill carre withcolor 0.8white ;
    fill carre transformed T withcolor white ;
    draw carre ; draw carre transformed T ;

    picture dessin ;
    dessin = currentpicture ;
    for i = 1 upto 8:
        dessin := dessin transformed T transformed T ;
        draw dessin ;
    endfor
endfig ;
```



1.7 Textes et graphiques

Les deux commandes `label` et `dotlabel` permettent d'écrire un texte à une position donnée, comme le montre l'exemple ci-dessous.

```
beginfig(6) ;
    z0 = (0,0) ; z1 = (3cm,0) ;

    label.top("top",z0) ; label.bot("bot",z0) ;
    label.lft("lft",z0) ; label.rt("rt",z0) ;

    labeloffset := 12pt ;

    dotlabel.ulft("ulft",z1) ; dotlabel.urt("urt",z1) ;
    dotlabel.llft("llft",z1) ; dotlabel.lrt("lrt",z1) ;

    labeloffset := 3bp ; % défaut
endfig ;
```



On aura noté la modification du paramètre `labeloffset`, qui a permis d'écarter les étiquettes du deuxième point.
Bien sûr, MetaPost reste complice de TEX : on peut remplacer toute occurrence d'une chaîne de caractère par une commande `btex ... etex`, où les trois points de suspension sont n'importe quelle commande TEX valide, ce qui est utlisé dans les exemples ci-dessous. La valeur d'une telle expression est en réalité du type `picture`.
Pour "mesurer" un texte, MetaPost offre cinq opérateurs qui s'appliquent d'ailleurs plus généralement à toute image, et pas seulement à un texte (composé par TEX ou pas), et que résume la figure suivante.

```
beginfig(7) ;
    picture texte ;

    label("mesure de texte" infont defaultfont scaled 5,(0,0) ;
    texte = currentpicture ;

    draw llcorner texte -- lrcorner texte
        -- urcorner texte -- ulcorner texte -- cycle ;
    label.llft("llcorner",llcorner texte) ;
    label.lrt("lrcorner",lrcorner texte) ;
    label.urt("urcorner",urcorner texte) ;
    label.ulft("ulcorner",ulcorner texte) ;
    % et on dispose aussi de center texte
    dotlabel.top("center",center texte) ;
endfig ;
```

1.8 Compléments sur les graphiques

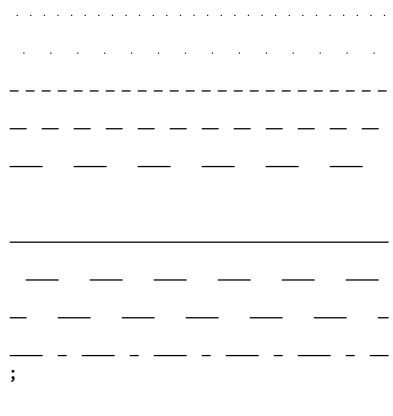
1.8.1 Pointillés

Pour tracer un chemin `p` en pointillés on utilise la commande

`draw p dashed ...`

où les `...` désigne un motif de pointillés.
Des motifs prédéfinis existent : à savoir `evenly` (traits de 3bp séparés par des espaces de même longueur), et `withdots` (points séparés par des espaces de 5bp).
Bien entendu, on peut modifier ces motifs par décalage et changement d'échelle, ou même créer de nouveaux motifs , comme indiqué ci-dessous.

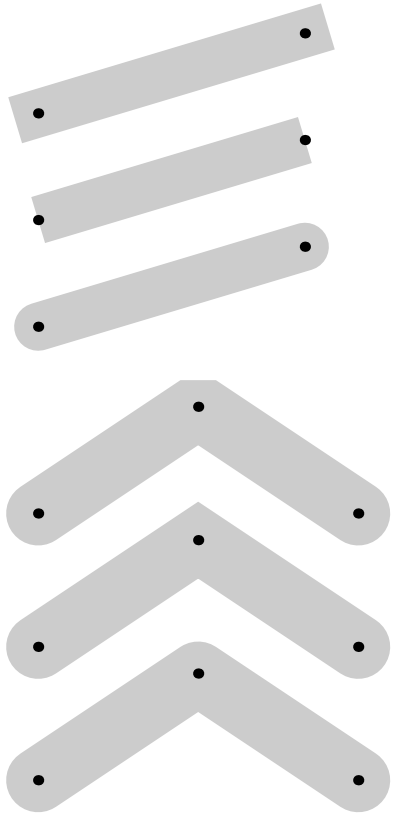
```
beginfig(8) ;
    path p ;
    pair d ;
    p = (0,0) -- (5cm,0) ;
    d = (0,-5mm) ;

    draw p             dashed withdots ;
    draw p shifted   d dashed withdots scaled 2 ;
    draw p shifted  2d dashed evenly ;
    draw p shifted  3d dashed evenly scaled 2 ;
    draw p shifted  4d dashed evenly scaled 4 ;

    draw p shifted  6d ;
    draw p shifted  7d dashed evenly scaled 4 shifted (6bp,0) ;
    draw p shifted  8d dashed evenly scaled 4 shifted (18bp,0) ;

    draw p shifted 9d dashed dashpattern(on 12bp off 6bp on 3bp off 6bp) ;
endfig ;
```



1.8.2 Options PostScript

Les réglages de PostScripts nommés `linecap` (par défaut : `rounded`), `linejoin` (par défaut : `rounded`), et `mitterlimit` (par défaut : 10) sont disponibles.

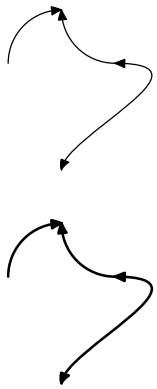
```
beginfig(9) ;
    for i=0 upto 2: z[i] = (0,40i) ; z[i+3]-z[i] = (100,30) ; endfor
    pickup pencircle scaled 18 ;
    linecap := rounded ; draw z0..z3 withcolor .8white ;
    linecap := butt ; draw z1..z4 withcolor .8white ;
    linecap := squared ; draw z2..z5 withcolor .8white ;
    linecap := rounded ; % défaut
    for i=0 upto 5: draw z[i] withpen pencircle scaled 4bp ; endfor

    for i=6 upto 8:
        z[i] = (0,50i) shifted (0,-470) ;
        z[i+3] - z[i] = (60,40) ; z[i+6] - z[i] = (120,0) ;
    endfor
    pickup pencircle scaled 24 ;
    linejoin := rounded ; draw  z6--z9--z12 withcolor .8white ;
    linejoin := mitered ; draw z7--z10--z13 withcolor .8white ;
    linejoin := beveled ; draw z8--z11--z14 withcolor .8white ;
    linejoin := rounded ; % défaut
    for i=6 upto 14: draw z[i] withpen pencircle scaled 4bp ; endfor
endfig ;
```



1.8.3 Flèches

MetaPost offre deux commandes très simples pour tracer des flèches, comme l'illustre l'exemple ci-dessous. Il y a d'autres réglages disponibles, que nous ne développerons pas ici davantage.
On observera que la taille de la flèche augmente avec celle de la plume de tracé. En outre, si l'on regarde de près, on verra que les flèches ne sont pas dessinées avec des lignes brisées, mais des courbes.

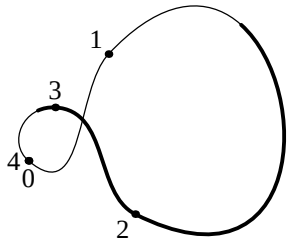
```
beginfig(10) ;
    drawarrow (0,0){up}..{right}(20,20) ;
    drawarrow reverse((20,20){down}..{right}(40,0)) ;
    drawdblarrow (40,0){right}..{down}(20,-40) ;

    pickup pencircle scaled 1bp ;
    drawarrow ((0,0){up}..{right}(20,20)) shifted (0,-80) ;
    drawarrow (reverse((20,20){down}..{right}(40,0))) shifted (0,-80) ;
    drawdblarrow ((40,0){right}..{down}(20,-40)) shifted (0,-80) ;
endfig ;
```



1.8.4 Paramétrisation des chemins

Un chemin `p` est un arc paramétré, le paramètre `t` variant de 0 à la "longueur" de l'arc, qu'on obtient avec la commande `length p`. Le point de paramètre `t` s'obtient alors facilement par la commande `point t of p`.
On peut facilement s'intéresser à un sous-arc correspondant aux variations $t \in [t_1, t_2]$ du paramètre grâce à la commande `subpath (t1,t2) of p`.

```
beginfig(11) ;
    path p ;
    p = (0,0) .. (30,40) .. (40,-20) .. (10,20) .. cycle ;
    draw p ;

    dotlabel.bot ("0",point 0 of p) ;
    dotlabel.ulft("1",point 1 of p) ;
    dotlabel.llft("2",point 2 of p) ;
    dotlabel.top ("3",point 3 of p) ;
    dotlabel.lft ("4",point 4 of p) ;

    draw subpath(1.3,3.2) of p withpen pencircle scaled 1.5bp ;
endfig ;
```

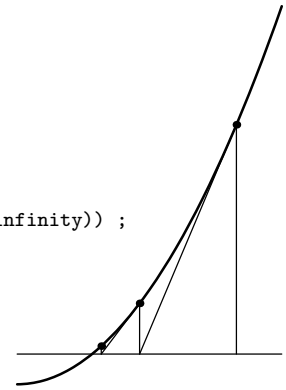


MetaPost nous permet aussi de déterminer une tangente en un point : la commande `direction t of p` renvoie un vecteur tangent au point de paramètre `t` de l'arc correspondant au chemin `p`.
Inversement, `directiontime (a,b) of p` renvoie le paramètre du premier point de l'arc où `(a,b)` dirige la tangente (et -1 s'il n'y a pas de tel point) et `directionpoint (a,b) of p` renvoie le point correspondant.
Si l'on s'intéresse à l'abscisse curviligne, on dispose de la commande `arclength`, et de la commande `arctime` définie ainsi : si `arctime a of p` vaut `t` c'est que l'on dispose de l'égalité

`arclength subpath (0,t) of p` = `a`.

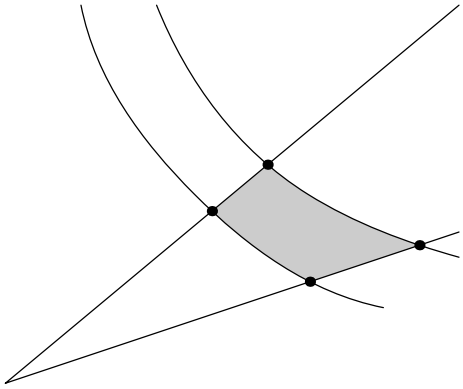
Si `p` et `q` sont deux chemins, `p intersectiontimes q` est un couple `(tp,tq)` de paramètres correspondants à un point d'intersection des deux chemins (s'ils ne se coupent pas, on obtient $(-1,-1)$). Dans le cas de plusieurs intersections, les règles utilisées pour choisir le point résultat sont complexes, et le lecteur est gentiment renvoyé au METAFONTbook. Notons l'existence de la commande `intersectionpoint` qui rend plutôt le point d'intersection lui-même.

```
beginfig(12) ;
    path courbe ;
    numeric t[] ;

    courbe = (0,-4mm)
        for i = 1 upto 10:
            .. (i*3.5mm,i*i*.5mm-4mm)
        endfor ;
    x1 = 29mm ;
    for i = 1 upto 3:
        (t[i],whatever) = courbe intersectiontimes ((x[i],-infinity)--(x[i],infinity)) ;
        z[i] = point t[i] of courbe ;
        z[i] - (x[i+1],0) = whatever * direction t[i] of courbe ;
        draw (x[i],0)--z[i]--(x[i+1],0) ;
        draw z[i] withpen pencircle scaled 3bp ;
    endfor
    draw (0,0) -- (35mm,0) ;
    draw courbe withpen pencircle scaled 1bp ;
endfig ;
```



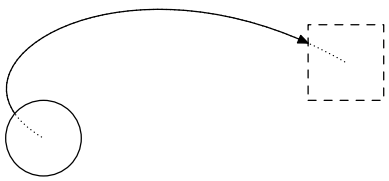
Pour continuer sur le même sujet, nous présentons un exemple d'utilisation de la commande `buildcycle` qui construit un chemin fermé à partir de plusieurs (pas forcément 4...) chemins qui se croisent, afin de pouvoir par la suite appliquer la commande `fill`, par exemple.

```
beginfig(13) ;
    path d[],p[],c ;
    d1 = (0,0)--(6cm,5cm) ;
    d2 = (0,0)--(6cm,2cm) ;
    p1 = (1cm,5cm){1,-5}..(2.5cm,2.5cm){2.5,-2.5}..(5cm,1cm){5,-1} ;
    p2 = (2cm,5cm){2,-5}..(4cm,2.5cm){4,-2.5}..(6cm,5/3 cm){18,-5} ;
    c = buildcycle(p1,d2,reverse p2,reverse d1) ;
    fill c withcolor .8white ;
    draw d1 ; draw d2 ; draw p1 ; draw p2 ;
    draw (p1 intersectionpoint d1) withpen pencircle scaled 4bp ;
    draw (p1 intersectionpoint d2) withpen pencircle scaled 4bp ;
    draw (p2 intersectionpoint d1) withpen pencircle scaled 4bp ;
    draw (p2 intersectionpoint d2) withpen pencircle scaled 4bp ;
endfig ;
```



Pour conclure, disons deux mots de deux commandes similaires, `cutbefore` et `cutafter`, qui sont d'un usage constant dans le dessin d'automates, comme nous le verrons ci-dessous.
Si `p` et `q` sont deux chemins qui se coupent, `p cutbefore q` renvoie le chemin `p` privé de sa partie qui se trouve avant le premier point d'intersection. On l'utilisera en particulier pour le dessin de flèches allant d'un objet à un autre, comme ci-dessous.

```
beginfig(14) ;
    path cercle,rectangle,p ;
    z0 = (0,0) ; z1 = (4cm,1cm) ;
    cercle = fullcircle scaled 1cm shifted z0 ;
    rectangle =
        ((-5mm,-5mm)--(5mm,-5mm)--(5mm,5mm)--(-5mm,5mm)--cycle)
        shifted z1 ;
    draw cercle ; draw rectangle dashed evenly ;
    p = z0{dir150}..z1{dir-30} ;
    draw p dashed withdots scaled 0.3 ;
    drawarrow (p cutbefore cercle cutafter rectangle) ;
endfig ;
```



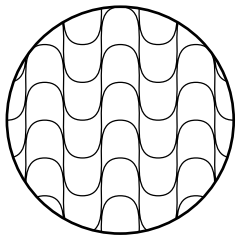
1.8.5 Images et pochoirs

Nous avons déjà parlé du type des images (`picture`). En fait chaque commande classique correspond à un ordre sur la variable `currentpicture`, comme indiqué dans le tableau ci-dessous, où `p` désigne un chemin, `c` un chemin fermé, et `pic` une image :

commande	équivalent
`draw pic`	`addto currentpicture also pic`
`draw p`	`addto currentpicture doublepath p withpen currentpen`
`fill c`	`addto currentpicture contour c`
`filldraw c`	`addto currentpicture contour c withpen currentpen`
`undraw pic`	`addto currentpicture also pic withcolor background`
`undraw p`	`addto currentpicture doublepath p withpen currentpen withcolor background`
`unfill c`	`addto currentpicture contour c withcolor background`
`unfilldraw c`	`addto currentpicture contour c withpen currentpen withcolor background`

Terminons par la commande `clip pic to c` qui permet, à la façon d'un pochoir, de ne conserver de l'image `pic` que ce qui est à l'intérieur du chemin fermé `c`, comme dans l'exemple ci-dessous.

```
beginfig(15) ;
    path p,q ;
    p = (0,-5mm){right}
        for i = 1 upto 10 :
            ..(i*5mm,((-1)**i) * 5mm){right}
        endfor ;
    for i = 0 upto 10 : draw p shifted (0,i*5mm) ; endfor
    q = fullcircle scaled 3cm shifted (4cm,4cm) ;
    clip currentpicture to q ;
    draw q ;
endfig ;
```



1.9 Programmation

1.9.1 Noms

On trouvera dans le manuel de référence de MetaPost une discussion approfondie et complète des noms de variables.
Contentons-nous ici de quelques éléments de base : un ⟨tag⟩ est un nom de variable "élémentaire" (MetaPost interdit à peu près seulement les mots-clés du langage). Une variable peut être une suite de *tag*(s) comme par exemple `f.bot` ou `a.b.c` où il ne faut voir dans le point (`.`) qu'un séparateur de lexèmes pour l'analyseur lexical de MetaPost.
Plus précisément, `x2r` se compose de trois lexèmes : le *tag* `x`, le nombre 2, le *tag* `r` ; `a.ba.c` se compose de trois lexèmes : le *tag* `a`, le *tag* `ba`, le *tag* `c`. La notation `x[i]r` se comprend alors, si par exemple `i`=2.72, comme la variable `x2.72r`, qui se compose de trois lexèmes : le *tag* `x`, le nombre 2.72, et le *tag* `r`.
Voici la syntaxe générale des variables :

⟨variable⟩ → ⟨tag⟩⟨suffixe⟩
⟨suffixe⟩ → ⟨vide⟩ | ⟨suffixe⟩⟨indice⟩ | ⟨suffixe⟩⟨tag⟩
⟨indice⟩ → ⟨nombre⟩ | `[`⟨expr. numérique⟩`]`

On a déjà dit qu'on peut déclarer des variables, et nous l'avons fait souvent. Une exception à ce propos : pour déclarer des variables indicées, on ne peut écrire `numeric q1, q2, q3, p3.4q, p4.5q ;` par exemple, mais on doit utiliser la déclaration `numeric q[], p[]q ;`

1.9.2 Structures de contrôle

Nous avons déjà donné de multiples exemples d'usage des boucles. La syntaxe générale est

`for`⟨identificateur⟩`=`⟨expression⟩ `step` ⟨expression⟩ `until` ⟨expression⟩ `:` ⟨corps de la boucle⟩ `endfor`

En fait le mot-clé que nous avons utilisé plus haut, à savoir `upto`, n'est qu'une abréviation pour `step 1 until`, et de même on dispose de l'abréviation `downto` pour `step -1 until`.
En relisant les exemples ci-dessus, on s'apercevra que le corps de la boucle peut-être constitué de n'importe quel texte, et pas seulement d'expressions valides complètes. Il faut, en programmation MetaPost, ou Metafont, se placer dans la tournure d'esprit de la programmation TEX, et penser que MetaPost "mange" mot à mot ce qu'on lui fait avaler.
D'autre part, on dispose de deux autres structures de boucles : le mot-clé `forever` introduit une boucle infinie, dont on sortira à l'aide d'un `exitif` ⟨expr. booléenne⟩ `;` ou bien d'un `exitunless` ⟨expr.booléenne⟩ `;`.
Ainsi, pourrait-on écrire en MetaPost un équivalent d'une boucle `while` en écrivant

`forever :` `exitunless` ⟨expr. booléenne⟩ `;` ⟨corps de la boucle⟩ `endfor`

Enfin, MetaPost propose la syntaxe

`forsuffixes` ⟨identificateur⟩ `=` ⟨liste de noms⟩ `:` ⟨corps de boucle⟩ `endfor`

où la ⟨liste de noms⟩ est une liste de noms de variables (de "suffixes") séparés par des virgules.
On dispose également d'expressions conditionnelles, dont la syntaxe s'écrit ainsi :

⟨cond⟩ → `if` ⟨expr. booléenne⟩ `:` ⟨clause⟩ ⟨alternative⟩ `fi`
⟨alternative⟩ → ⟨vide⟩| `else :` ⟨clause⟩ | `elseif` ⟨expr. booléenne⟩ `:` ⟨clause⟩ ⟨alternative⟩

1.9.3 Définitions de macros

Remarque préliminaire : ceci n'est pas une présentation exhaustive...
Les macros s'écrivent facilement en MetaPost, comme par exemple :

```
def fill = addto currentpicture contour enddef ;
def rotatedaround(expr z, d) = shifted -z rotated d shifted z enddef ;
```

le deuxième exemple montrant comment introduire des paramètres : le mot-clé `expr` signifie que les arguments `z` et `d` doivent être des expressions MetaPost valides, de types quelconques (d'ailleurs ici `z` est censé être du type `pair` et `d` du type `numeric`).

Blocs et variables locales On dispose d'une structure de bloc, encadré par les mots-clés `begingroup` et `endgroup`, qui permet surtout de définir des variables locales.
Il existe deux types de blocs : les blocs-procédures et les blocs-fonctions, pour reprendre une terminologie *à la Pascal*.

⟨bloc⟩ → `begingroup` ⟨suite d'instructions⟩ `endgroup` | `begingroup` ⟨suite d'instructions⟩ ⟨expression⟩ `endgroup`

Pour créer une variable locale, on utilise le mot-clé `save` juste après le `begingroup` concerné. Ainsi, par exemple, peut-on définir

```
def whatever = begingroup save x ; x endgroup ;
```

Expressions en paramètres Comme MetaPost ne fait aucun contrôle de typage pour un argument annoncé par `expr`, il peut être intéressant d'utiliser les tests de type déjà décrits plus haut. Par exemple, la fonction `milieu` suivante fonctionnera aussi bien avec un argument chemin ou image :

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

Une autre version incompréhensible mais correcte de cette fonction serait la suivante :

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

Paramètres suffixes et textuels À la place du mot-clé `expr`, MetaPost nous propose également `suffix` et `text` pour annoncer les paramètres de la macro.
Le premier correspond à des noms de variables, le second à n'importe quelle suite de lexèmes.
Par exemple, il existe une commande `hide` qui se contente d'avaler et d'exécuter ce qu'on lui fournit en argument, sans rien renvoyer. Ainsi, `show hide(numeric a,b ; a+b = 3 ; a-b = 1) a ;` renvoie-t-il 2 (`show` est la commande qui permet, à toutes bonnes fins de déverminage, d'afficher dans le *log-file* des valeurs d'expressions MetaPost).
On pourrait écrire ainsi la macro `hide` :

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

Les paramètres annoncés par `suffix` permettent en fait un passage par référence des arguments : après avoir défini

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

il est tout à fait légitime de demander par exemple `incr(a32b) ;`.
Remarque : dans un appel de macro, il est en général indifférent d'utiliser `,` ou `)(`. Ainsi, si une macro est définie par `def foo(expr a)(suffix b)=...`, on pourra l'appeler aussi bien par une commande comme `foo(2,x)` que par `foo(2)(x)`.
Là où cela se corse, c'est quand un argument est du type `text` : il devient obligatoire d'utiliser la syntaxe `)(` : si je définis `def foo(text t)(expr b)=...`, l'appel `foo(1,2,3)(4)` est correct (et à `t` sera substitué `1,2,3`), mais la commande `foo(1,2,3,4)` est incomplète : il manque la valeur du paramètre `b`.

Les macros "vardef" Une macro définie par `vardef` présente quelques particularités. Ce qui la distingue d'abord (d'une macro définie par `def`), c'est qu'un bloc `begingroup`–`endgroup` encadre implicitement ses commandes.
D'autre part, le nom d'une macro `vardef` n'est pas limité à un identificateur simple (un "*tag*") — dans ce cas seule l'existence implicite du bloc `begingroup`–`endgroup` fait la différence. Ici, au contraire, le nom d'une macro peut être un nom indicé.
Ainsi, si je définis `vardef a[]bc(expr x) = ... enddef ;`, je pourrais l'appeler par `a2bc(x)` ou par `a3bc(x)`. Bien sûr, il faut pouvoir récupérer l'indice utilisé dans l'appel. C'est le rôle de deux arguments implicites (de type `suffix`) qui sont nommés `@` et `#@`.
Dans l'appel `a2bc(x)`, `#@` désigne `a2` et `@` désigne `bc` : `@` désigne le dernier *tag* qui compose le nom de macro figurant dans l'appel, et `#@` désigne tout ce qui précède.
Une autre syntaxe fort pratique est la suivante :

`vardef` ⟨tag⟩`@# (`⟨paramètres⟩`) =` ⟨corps de la macro⟩ `enddef ;`

qui permet d'assigner au paramètre implicite `@#` tout le suffixe qui suit le *tag* initial dans le nom de l'appel.
Par exemple, l'abréviation des couples $(x.?, y.?)$ en $z.?$ se définit en une seule fois par

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

qui permet tout aussi bien d'écrire `z1` (et ici `@#` désigne `1`) que par exemple `z.a3` (et ici `@#` désigne `a3`).
On trouvera un exemple dans le dessin d'automate de la dernière partie, qui définit :

```
vardef miArete(suffix a,b)(expr p) =
drawarrow p cutbefore bpath.a cutafter bpath.b ;
point .5*length p of p
enddef ;

vardef miBoucle@# (expr p) =
    miArete(@#,@#)(@#.c{curl0}..@#.c+p..{curl0}@#.c) enddef ;
```

Un appel comme `miBoucle.ab(z0)` sera expansé en :

```
drawarrow ab.c{curl0}..ab.c+z0..{curl0}ab.c cutbefore bpath.ab cutafter bpath.ab ;
point .5*length(ab.c{curl0}..ab.c+z0..{curl0}ab.c) of (ab.c{curl0}..ab.c+z0..{curl0}ab.c)
```

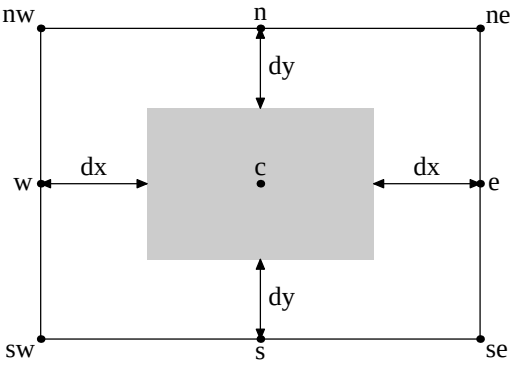
2 Le fichier de macros `boxes`

Le fichier `boxes.mp` est fourni avec la distribution standard de MetaPost, il définit différentes macros qui permettent un usage facile de boîtes entourant tout type d'objets graphiques, et est d'un usage constant quand on veut dessiner des arbres ou des automates.
Pour incorporer ses définitions, il suffit de la commande `input boxes ;`.

2.1 Boîtes rectangulaires

Pour créer une boîte entourant un objet graphique, il suffit d'écrire `boxit.`⟨suffixe⟩`(`⟨objet⟩`)` : cela définit une boîte rectangulaire de nom ⟨suffixe⟩ qui encadre l'⟨objet⟩ graphique. Pour dessiner la boîte (et son contenu), il suffit d'utiliser la commande `drawboxed(`⟨liste de noms de boîtes⟩`)`.
On peut récupérer le rectangle qui entoure la boîte de nom `a` par `bpath.a`. Le centre de la boîte est simplement `a.c`, et on obtient les huit points cardinaux par `a.n`, `a.nw`, `a.w`, `a.sw`, `a.s`, `a.se`, `a.e`, `a.ne`. En fait la boîte `a` laisse un espace autour de l'objet, que mesurent les quantités `a.dx` et `a.dy` (les valeurs par défaut — bien sûr modifiables — sont `dxdefault` et `dydefault`).

```
beginfig(16) ;
    fill (0,0)--(3cm,0)--(3cm,2cm)--(0,2cm)--cycle withcolor 0.8white ;
    picture gris ;
    gris = currentpicture ;
    currentpicture := nullpicture ;

    % pour y voir quelque chose ...
    defaultdx := 40pt ; defaultdy := 30pt ;

    boxit.a(gris) ;
    a.c = (0,0) ;
    drawboxed(a) ;
    dotlabel.top ("n", a.n) ;
    dotlabel.ulft("nw",a.nw) ;
    dotlabel.lft ("w", a.w) ;
    dotlabel.llft("sw",a.sw) ;
    dotlabel.bot ("s", a.s) ;
    dotlabel.lrt ("se",a.se) ;
    dotlabel.rt  ("e", a.e) ;
    dotlabel.urt ("ne",a.ne) ;
    dotlabel.top ("c", a.c) ;
    drawdblarrow a.w -- a.w shifted( a.dx,0) ;
    drawdblarrow a.e shifted(-a.dx,0) -- a.e ;
    drawdblarrow a.s -- a.s shifted(0, a.dy) ;
    drawdblarrow a.n shifted(0,-a.dy) -- a.n ;
    label.top("dx",a.w shifted ( a.dx/2,0)) ;
    label.top("dx",a.e shifted (-a.dx/2,0)) ;
    label.rt ("dy",a.s shifted (0, a.dy/2)) ;
    label.rt ("dy",a.n shifted (0,-a.dy/2)) ;

    defaultdx := 3bp ; defaultdy := 3bp ; % défaut
endfig ;
```



Dans l'exemple précédent, nous avons placé la boîte grâce à l'équation `a.c = (0,0) ;`. En général, si plusieurs boîtes doivent être juxtaposées, les calculs nécessaires seraient fastidieux. C'est pourquoi on dispose de la commande `boxjoin(`⟨équations⟩`)` qui permet de faciliter le processus en expliquant comment deux boîtes consécutives doivent être juxtaposées : il suffira alors de placer la première boîte. La convention utilisée par `boxjoin` est d'appeler `a` la première boîte et `b` la seconde, et de donner des équations comme `boxjoin(a.se = b.sw ; a.ne = b.nw) ;`, par exemple, pour aligner les boîtes horizontalement de gauche à droite.

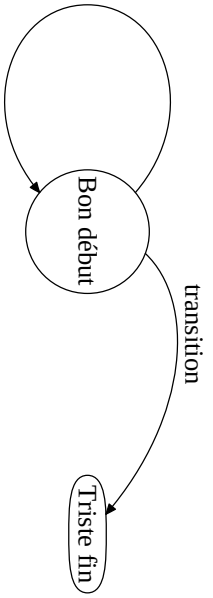
Si l'on veut commencer un nouveau bloc de boîtes dans un autre alignement, il suffira de donner une nouvelle commande `boxjoin`.

```
beginfig(17) ;
    boxjoin(a.sw = b.nw ; a.se = b.ne) ;
    boxit.a("A") ; boxit.b("B") ;
    boxit.c("C") ; boxit.d("D") ;
    a.c = (0,0) ;
    drawboxed(a,b,d,c,d) ;
endfig ;
```

2.2 Boîtes rondes

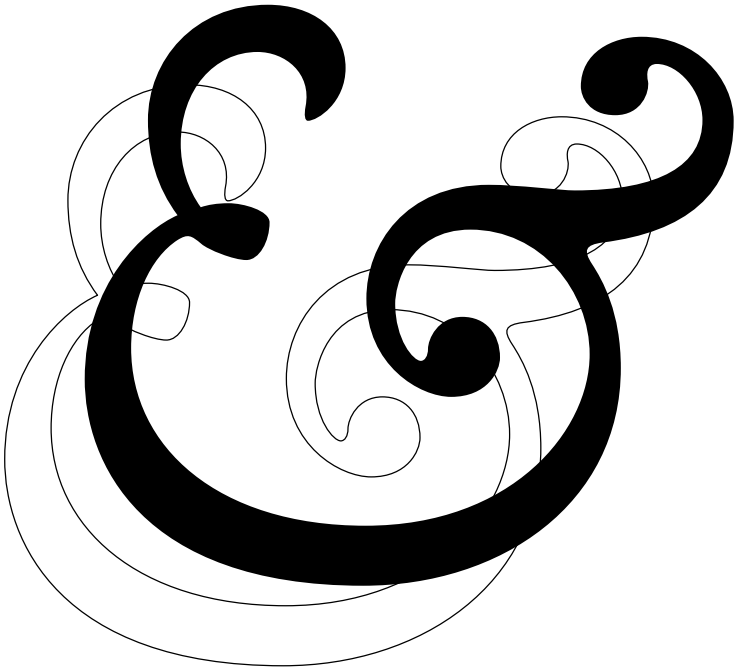
Les boîtes rondes s'obtiennent de façon analogue, en utilisant `circleit` au lieu de `boxit`. La seule vraie différence est que cette fois, après une commande `circleit.a(...)` ;, sont définis `a.c`, et seulement les quatre points cardinaux `a.n`, `a.w`, `a.s`, `a.e`, mais pas `a.ne`, etc.
En outre, si l'on veut non pas un cercle mais une ellipse, on imposera `a.dx = a.dy`, puisque par défaut les longueurs `dx` et `dy` sont calculées de sorte qu'on obtienne un cercle.

```
beginfig(18) ;
    circleit.a("Bon début") ;
    % je veux un vrai cercle

    circleit.b("Triste fin") ;
    % je veux un ovale
    b.dx = b.dy ;

    a.c = (0,0) ; b.c = (4cm,0) ;
    drawboxed(a,b) ;

    path p ;
    p = a.c{up}..b.c{dir-45} ;
    drawarrow p cutbefore bpath.a cutafter bpath.b ;
    label.top("transition",point 0.5 of p) ;

    path q ;
    q = a.c{dir120}..a.c shifted (-3cm,0)..a.c{dir60} ;
    drawarrow q cutbefore bpath.a cutafter bpath.a ;

    picture dessin ;
    dessin = currentpicture ;
    currentpicture := nullpicture ;

    draw dessin rotated -90 ;
endfig ;
```



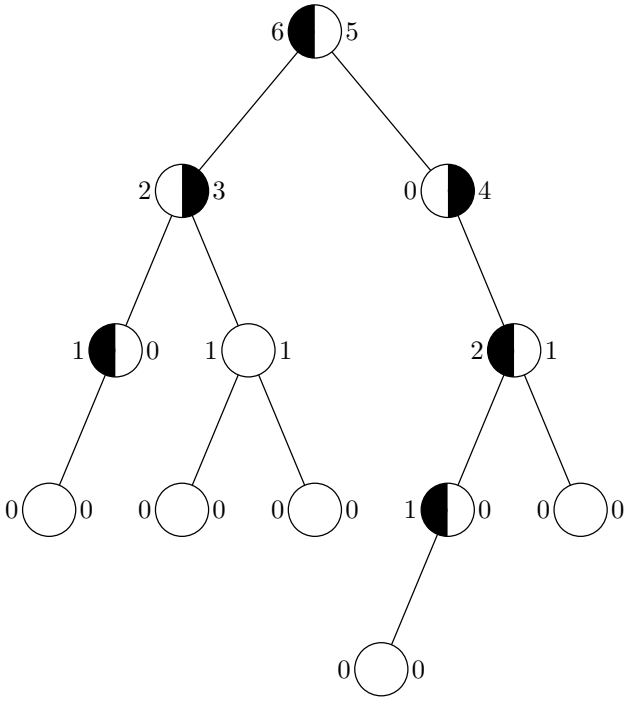
3 Quelques exemples

3.1 Un dessin de caractère



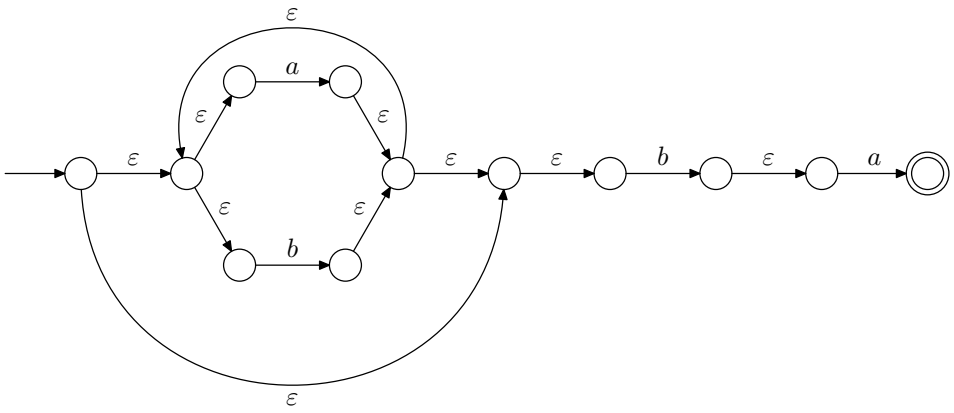
```
prologues := 2 ;

beginfig(1)

    path p ;

    z1 = (241.0005, 514.0508) ;
    z2 = (241.0005, 550.0513) ;
    z3 = (263.5005, 570.1514) ;
    z4 = (275.8008, 575.5513) ;

    z5 = (268.0005, 586.0513) ;
    z6 = (264.7007, 597.7515) ;
    z7 = (264.7007, 611.252) ;

    z8 = (264.7007, 633.4521) ;
    z9 = (281.8008, 654.4521) ;
    z10 = (309.7012, 654.4521) ;

    z11 = (326.2012, 654.4521) ;
    z12 = (338.8018, 645.4521) ;
    z13 = (338.8018, 630.752) ;

    z14 = (338.8018, 617.8516) ;
    z15 = (328.3008, 610.9517) ;
    z16 = (324.7012, 610.9517) ;

    z17 = (323.501, 610.9517) ;
    z18 = (323.2012, 613.0518) ;
    z19 = (323.8008, 616.3516) ;

    z20 = (326.2012, 628.9517) ;
    z21 = (316.001, 636.752) ;
    z22 = (305.8008, 636.752) ;

    z23 = (289.6006, 636.752) ;
    z24 = (277.0005, 622.6519) ;
    z25 = (277.0005, 602.2515) ;

    z26 = (277.0005, 593.8516) ;
    z27 = (279.7007, 585.1514) ;
    z28 = (284.501, 578.5513) ;

    z29 = (287.2007, 579.4512) ;
    z30 = (290.501, 580.0513) ;
    z31 = (295.001, 580.0513) ;

    z32 = (300.4014, 580.0513) ;
    z33 = (310.3008, 577.3516) ;
    z34 = (310.3008, 572.8516) ;

    z35 = (310.3008, 565.6514) ;
    z36 = (306.4014, 558.751) ;
    z37 = (301.6006, 558.751) ;

    z38 = (296.8008, 558.751) ;
    z39 = (287.2007, 562.6514) ;
    z40 = (284.8008, 564.7515) ;

    z41 = (280.9009, 568.0513) ;
    z42 = (279.7007, 568.6514) ;
    z43 = (275.8008, 566.2515) ;

    z44 = (265.6006, 559.6514) ;
    z45 = (258.4004, 544.0508) ;
    z46 = (258.4004, 525.751) ;

    z47 = (258.4004, 488.8506) ;
    z48 = (290.8008, 459.1504) ;
    z49 = (346.3018, 459.1504) ;

    z50 = (401.2021, 459.1504) ;
    z51 = (430.3018, 494.8506) ;
    z52 = (430.3018, 523.3506) ;

    z53 = (430.3018, 547.0513) ;
    z54 = (412.002, 570.1514) ;
    z55 = (385.6016, 570.1514) ;

    z56 = (361.9014, 570.1514) ;
    z57 = (357.4014, 548.5513) ;
    z58 = (357.4014, 542.5508) ;

    z59 = (357.4014, 528.751) ;
    z60 = (364.3018, 520.9507) ;
    z61 = (367.002, 520.9507) ;

    z62 = (368.502, 520.9507) ;
    z63 = (369.7012, 522.751) ;
    z64 = (369.7012, 525.4507) ;

    z65 = (369.7012, 529.0508) ;
    z66 = (373.3018, 537.4512) ;
    z67 = (382.6016, 537.4512) ;

    z68 = (391.3018, 537.4512) ;
    z69 = (396.7021, 531.1509) ;
    z70 = (396.7021, 522.1509) ;

    z71 = (396.7021, 517.0508) ;
    z72 = (391.9014, 507.4507) ;
    z73 = (378.4014, 507.4507) ;

    z74 = (366.7012, 507.4507) ;
    z75 = (346.6016, 519.1509) ;
    z76 = (346.6016, 544.3511) ;

    z77 = (346.6016, 564.4512) ;
    z78 = (361.002, 586.9517) ;
    z79 = (392.502, 586.9517) ;

    z80 = (403.9023, 586.9517) ;
    z81 = (419.2021, 584.8516) ;
    z82 = (424.6025, 584.8516) ;

    z83 = (457.0029, 584.8516) ;
    z84 = (472.6025, 594.4517) ;
    z85 = (472.6025, 611.252) ;

    z86 = (472.6025, 622.0518) ;
    z87 = (463.6025, 632.252) ;
    z88 = (455.5029, 632.252) ;

    z89 = (451.6025, 632.252) ;
    z90 = (451.6025, 628.3521) ;
    z91 = (452.2021, 625.6519) ;

    z92 = (452.8027, 621.752) ;
    z93 = (449.8027, 613.0518) ;
    z94 = (439.9023, 613.0518) ;

    z95 = (430.002, 613.0518) ;
    z96 = (427.002, 619.9517) ;
    z97 = (427.002, 623.8521) ;

    z98 = (427.002, 636.4521) ;
    z99 = (438.7021, 642.4521) ;
    z100 = (449.8027, 642.4521) ;

    z101 = (470.2031, 642.4521) ;
    z102 = (484.3027, 626.8521) ;
    z103 = (484.3027, 610.9517) ;

    z104 = (484.3027, 583.0513) ;
    z105 = (465.7021, 569.2515) ;
    z106 = (435.4023, 565.3511) ;

    z107 = (428.502, 564.4512) ;
    z108 = (427.9023, 562.3511) ;
    z109 = (431.2021, 557.251) ;

    z110 = (435.1025, 551.251) ;
    z111 = (442.002, 539.251) ;
    z112 = (442.002, 518.251) ;

    z113 = (442.002, 467.25) ;
    z114 = (397.002, 436.6499) ;
    z115 = (345.4014, 436.6499) ;

    z116 = (261.7007, 436.6499) ;
    z117 = (241.0005, 483.4502) ;

    p  = for i=0 upto 38:
            z[3i+1] ..
                controls z[3i+2]
                and z[3*i+3] ..
         endfor
         cycle ;

    draw p ;
    fill p shifted (30,30) ;

endfig ;

end
```

3.2 Un arbre



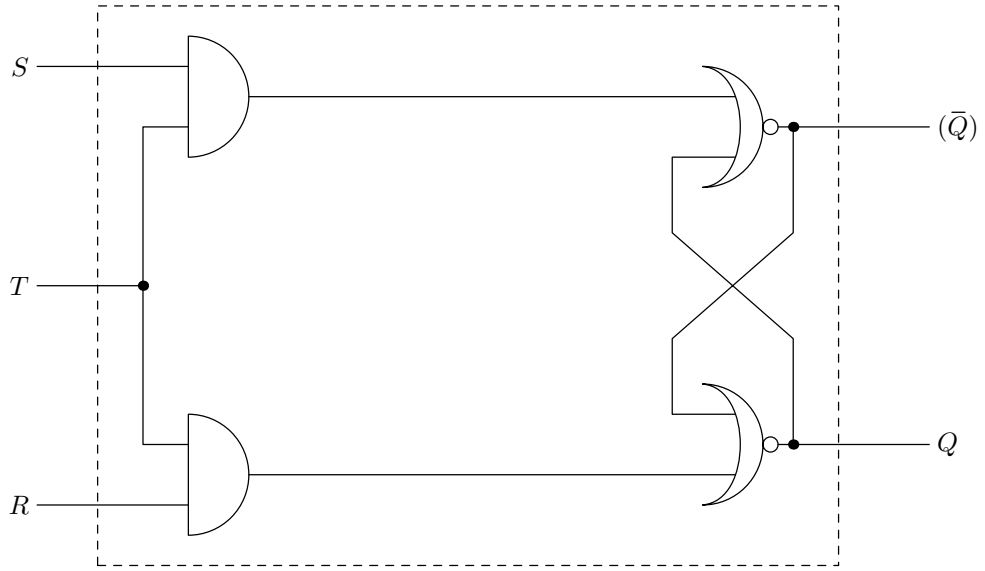
```
prologues := 2 ;
defaultfont := "CMR10" ;

beginfig(1)

    diametre = 20 pt ;
    h = 50 pt ;
    v = 60 pt ;

    path demigauche,demidroit ;
    demidroit  = halfcircle scaled diametre rotated -90 -- cycle ;
    demigauche = demidroit xscaled -1 ;

    picture bb,bn,nb ;
    fill fullcircle scaled diametre withcolor white ;
    draw fullcircle scaled diametre ;
    bb := currentpicture ;

    currentpicture := nullpicture ;
    fill demidroit withcolor black ;
    fill demigauche withcolor white ;
    draw fullcircle scaled diametre ;
    bn := currentpicture ;

    currentpicture := nullpicture ;
    fill demigauche withcolor black ;
    fill demidroit withcolor white ;
    draw fullcircle scaled diametre ;
    nb := currentpicture ;

    currentpicture := nullpicture ;

% définition des coordonnées des sommets de l'arbre
    z0 = (0,0) ;
    y0 - y1 = y1 - y3 = y3 - y6 = y9 - y11 = v ; y1 = y2 ; y3 = y4 = y5 ; y6 = y7 = y8 = y9 = y10 ;
    x10 - x9 = x9 - x8 = x8 - x7 = x7 - x6 = x4 - x3 = h ;
    x4 = 1/2[x7,x8] ; x5 = 1/2[x9,x10] ; x1 = 1/2[x3,x4] ; x0 = 1/2[x1,x2] ;
    x5 - x2 = x9 - x11 = h/2 ;
% tracé des arètes
    draw z0 -- z1 -- z3 -- z6 ;
    draw z1 -- z4 -- z8 ;
    draw z4 -- z7 ;
    draw z0 -- z2 -- z5 -- z10 ;
    draw z5 -- z9 -- z11 ;
% tracé des sommets
    def sbb(text g)(expr a)(text d) =
        draw bb shifted a ;
        label(g,a - (14pt,0)) ; label(d,a + (14pt,0)) ;
    enddef ;

    def sbn(text g)(expr a)(text d) =
        draw bn shifted a ;
        label(g,a - (14pt,0)) ; label(d,a + (14pt,0)) ;
    enddef ;

    def snb(text g)(expr a)(text d) =
        draw nb shifted a ;
        label(g,a - (14pt,0)) ; label(d,a + (14pt,0)) ;
    enddef ;

    snb("6")(z0)("5") ; sbn("2")(z1)("3") ; sbn("0")(z2)("4") ; snb("1")(z3)("0") ;
    sbb("1")(z4)("1") ; snb("2")(z5)("1") ; sbb("0")(z6)("0") ; sbb("0")(z7)("0") ;
    sbb("0")(z8)("0") ; snb("1")(z9)("0") ; sbb("0")(z10)("0") ; sbb("0")(z11)("0") ;
endfig ;

end
```

3.3 Un automate



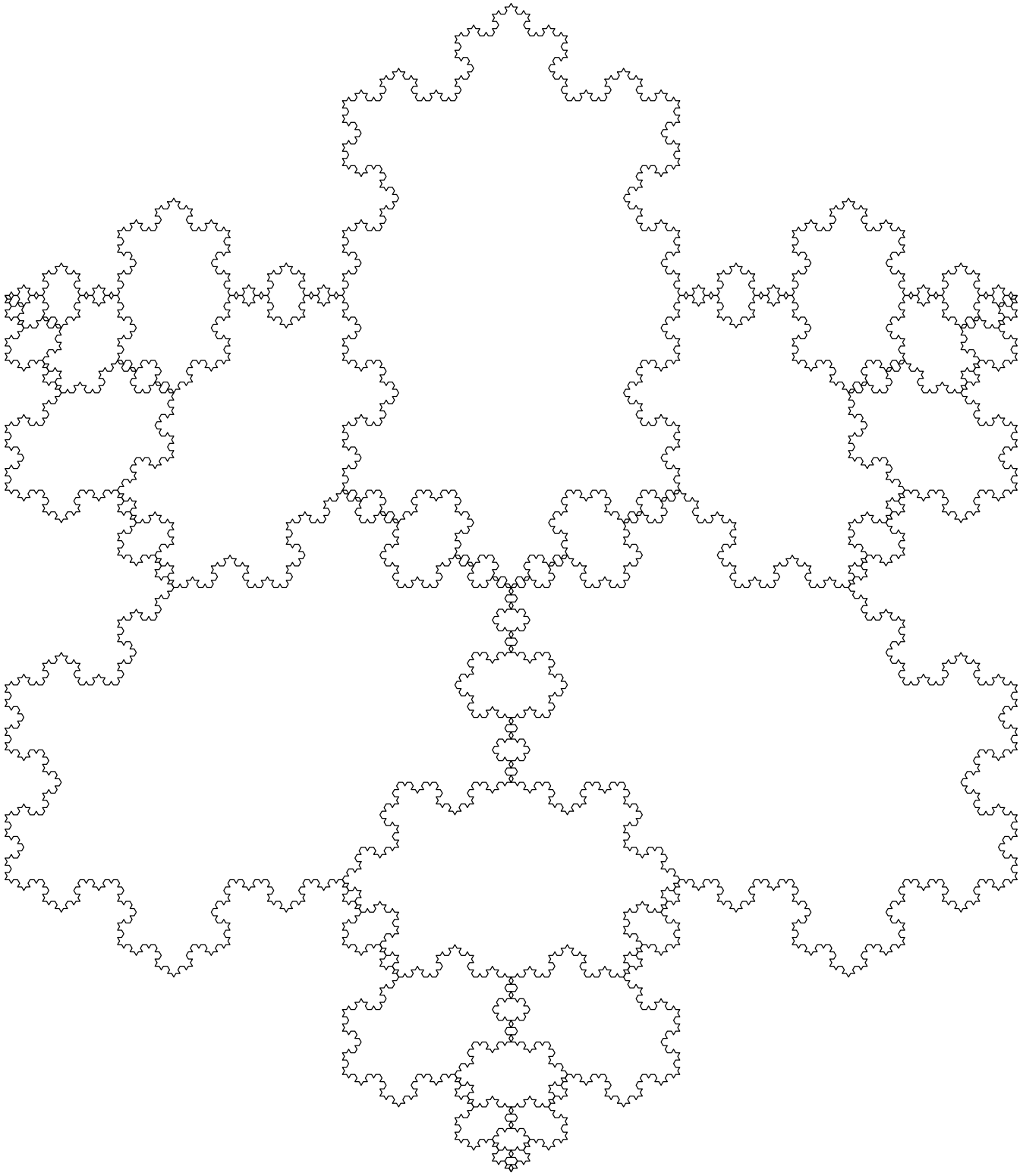
```
input boxes ;
prologues := 2 ;

vardef miArete(suffix a,b) expr p =
    drawarrow p cutbefore bpath.a cutafter bpath.b ;
    point .5*length p of p
enddef ;

vardef miBoucle@# expr p = miArete(@#,@#) @#.c{curl0}..@#.c+p..{curl0}@#.c enddef ;

def cercle(suffix a,b) = circleit.a() ; a.c = z.b ; enddef ;

beginfig(1)

     interim circmargin := 6bp ;

     z0 = (0,0) ;
     z1 - z0 = z7 - z6 = z8 - z7 = z9 - z8
              = z10 - z9 = z11 - z10 = z3 - z2 = z5 - z4
              = (z2 - z1) rotated -60 = (z4 - z1) rotated 60 = (z6 - z3) rotated 60
              = (14mm,0) ;

     cercle(a)(0) ; cercle(b)(1) ; cercle(c)(2) ; cercle(d)(3) ; cercle(e)(4) ; cercle(f)(5) ;
     cercle(g)(6) ; cercle(h)(7) ; cercle(i)(8) ; cercle(j)(9) ; cercle(k)(10) ; cercle(l)(11) ;

     drawboxed(a,b,c,d,e,f,g,h,i,j,k,l) ;

     interim circmargin := 8bp ;
     circleit.ll(pic l) ; ll.c = z11 ;
     drawboxed(ll) ;

     drawarrow (-1cm,0)--z0 cutafter bpath.a ;

     label.top(btex $\varepsilon$ etex,miArete(a,b) a.c--b.c) ;
     label.ulft(btex $\varepsilon$ etex,miArete(b,c) b.c--c.c) ;
     label.top(btex $a$ etex,miArete(c,d) c.c--d.c) ;
     label.urt(btex $\varepsilon$ etex,miArete(d,g) d.c--g.c) ;
     label.urt(btex $\varepsilon$ etex,miArete(b,e) b.c--e.c) ;
     label.top(btex $b$ etex,miArete(e,f) e.c--f.c) ;
     label.ulft(btex $\varepsilon$ etex,miArete(f,g) f.c--g.c) ;
     label.top(btex $\varepsilon$ etex,miArete(g,h) g.c--h.c) ;
     label.top(btex $\varepsilon$ etex,miArete(h,i) h.c--i.c) ;
     label.top(btex $b$ etex,miArete(i,j) i.c--j.c) ;
     label.top(btex $\varepsilon$ etex,miArete(j,k) j.c--k.c) ;
     label.top(btex $a$ etex,miArete(k,ll) k.c--l.c) ;
     label.top(btex $\varepsilon$ etex,miArete(g,b) g.c{dir72}..b.c) ;
     label.bot(btex $\varepsilon$ etex,miArete(a,h) a.c{down}..h.c) ;

endfig ;

end
```

3.4 Un circuit électronique



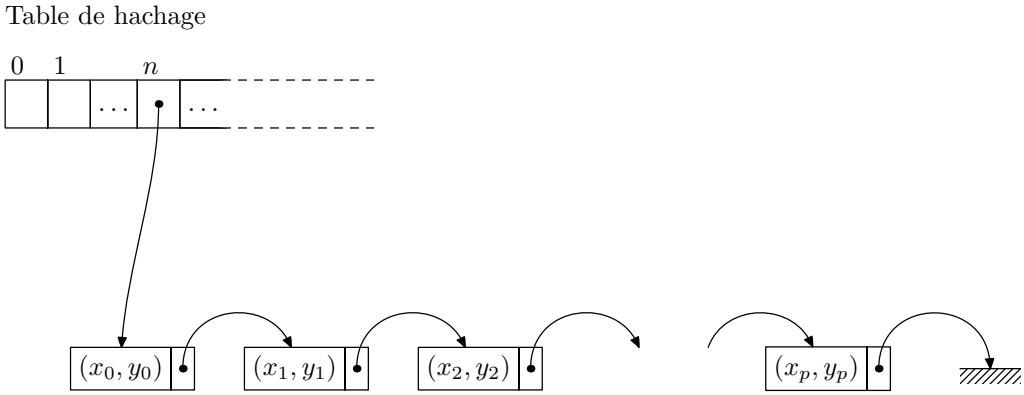
```
prologues := 2 ;

beginfig(1)

    def porteAND(expr a) =
        ((halfcircle scaled 16mm) -- cycle) rotated -90 shifted (-8mm,0) shifted a
    enddef ;

    def porteNOR(expr a) =
        begingroup
            save $ ;
            pair $ ;
            $ = a shifted (-2mm,0) ;
            ${up}..{left}($ shifted (-8mm,8mm))
                &($ shifted (-8mm,8mm)){dir -5}..($ shifted (-3mm,0)){down}..{dir 185}($ shifted (-8mm,-8mm))
                &($ shifted (-8mm,-8mm)){right}..{up}$
                &(${down}..($ shifted (1mm,-1mm)){right}..($ shifted (2mm,0)){up}..($ shifted (1mm,1mm)){left}..cycle)
        endgroup
    enddef ;

    def point(expr a) = draw a withpen pencircle scaled 4bp enddef ;

    z0 = (0,0) ; x1 = x0 ;
    x2 = x3 ; y2 = y0 - 4mm ; y3 = y1 + 4mm ;
    x2 - x0 = 7cm ; y0 - y1 = 5cm ;

    path p[] ;
    p0 = porteAND(z0) ; p1 = porteAND(z1) ;
    p2 = porteNOR(z2) ; p3 = porteNOR(z3) ;
    draw p0 ; draw p1 ; draw p2 ; draw p3 ;

    draw ((-2cm,0)--(0,0)) shifted (z0 + (-8mm,4mm)) ; draw ((-2cm,0)--(0,0)) shifted (z1 + (-8mm,-4mm)) ;
    label.lft(btex $S$ etex,z0 + (-28mm,4mm)) ; label.lft(btex $R$ etex,z1 + (-28mm,-4mm)) ;

    draw ((-28mm,0)--(-14mm,0)) shifted 1/2[z0,z1] ; point((-14mm,0) shifted 1/2[z0,z1]) ;
    label.lft(btex $T$ etex,1/2[z0,z1] + (-28mm,0)) ;

    draw ((-8mm,-4mm)--(-14mm,-4mm)--(-14mm,-46mm)--(-8mm,-46mm)) shifted z0 ;

    draw ((0,0)--(2cm,0)) shifted z2 ; draw ((0,0)--(2cm,0)) shifted z3 ;
    label.rt(btex $(\bar Q)$ etex,z2 + (2cm,0)) ; label.rt(btex $Q$ etex,z3 + (2cm,0)) ;
    point(z2 + (2mm,0)) ; point(z3 + (2mm,0)) ;

    x4 = x5 = x6 = x7 = x2 - 14mm ; x8 = x9 = x2 + 2mm ;
    y4 = y2 - 4mm ; y7 = y3 + 4mm ; y5 = y8 ; y6 = y9 ; y8=1/3[y2,y3] ; y9 = 2/3[y2,y3] ;
    draw (z2 + (2mm,0))--z8--z6--z7--(z3 + (-2mm, 4mm)) cutafter (reverse p3) ;
    draw (z3 + (2mm,0))--z9--z5--z4--(z2 + (-2mm,-4mm)) cutafter p2 ;

    draw z0--(z2 + (-2mm, 4mm)) cutafter (reverse p2) ;
    draw z1--(z3 + (-2mm,-4mm)) cutafter p3 ;

    draw (z0 + (-20mm, 12mm))--(z1 + (-20mm,-12mm))--(z3 + (8mm,-16mm))--(z2 + (8mm,16mm))--cycle dashed evenly ;

endfig ;

end
```

3.5 Une fractale



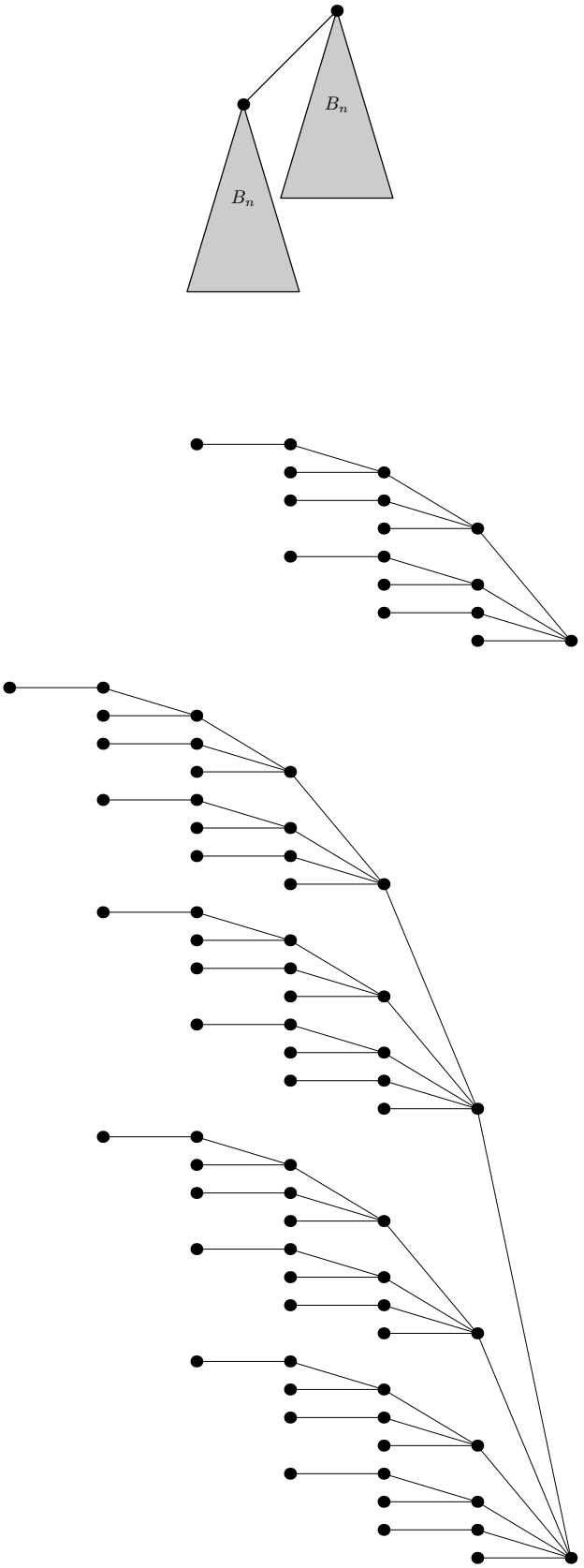
```
prologues := 2 ;
defaultfont := "CMR10" ;

beginfig(1)

    numeric l ;

    l = 500 ;

    z0 = (0,0) ; z4 = (l,0) ;
    z1 = 1/3[z0,z4] ; z3 = 2/3[z0,z4] ;
    z2 - z1 = (z3 - z1) rotated 60 ;

    def next(expr p) =
        (p scaled 1/3) -- (p scaled 1/3 rotated 60 shifted z1)
          -- (p scaled 1/3 rotated -60 shifted z2) -- (p scaled 1/3 shifted z3)
    enddef ;

    path etoile ;

    etoile = (0,0) -- (l,0) ;

    for i = 1 upto 5 :
        etoile := next(etoile) ;
    endfor

    z4 - z0 = (z5 - z0) rotated 60 ;

    draw etoile -- (etoile rotated -120 shifted z4) -- (etoile rotated 120 shifted z5) ;
    draw (etoile yscaled -1) -- (etoile yscaled -1 rotated -120 shifted z4)
                   -- (etoile yscaled -1 rotated 120 shifted z5) ;

endfig ;

end
```

3.6 Un schéma pour une table de hachage



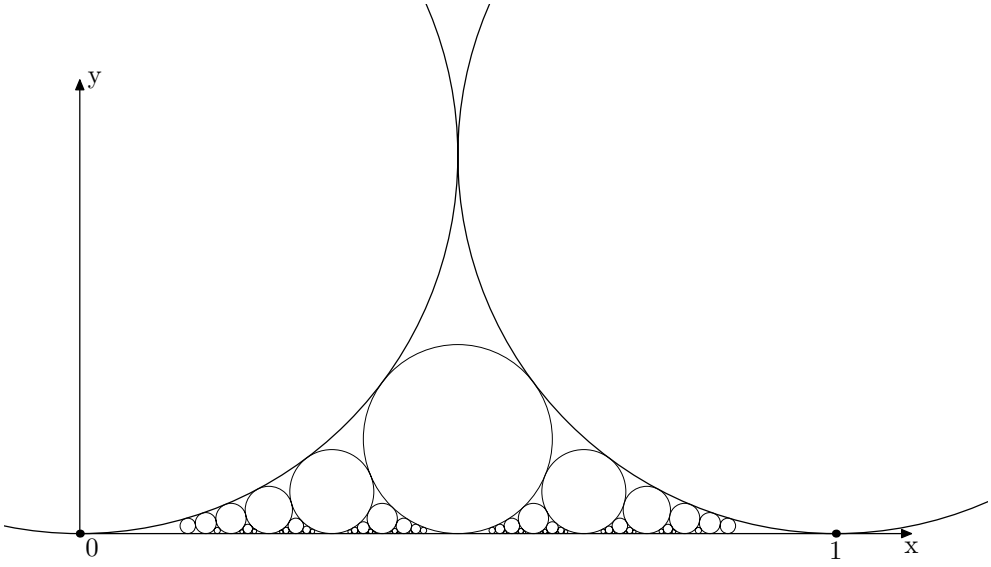
```
prologues := 2 ;
defaultfont := "CMR10" ;

input boxes ;

beginfig(1)
    boxjoin(a.se = b.sw ; a.ne = b.nw) ; % a et b sont les noms conventionnels de deux boîtes consécutives
    boxit.zero  (btex \strut\quad etex) ;
    boxit.un    (btex \strut\quad etex) ;
    boxit.deux  (btex \strut $\ldots$ etex) ;
    boxit.trois (btex \strut\quad etex) ;
    boxit.quatre(btex \strut $\ldots$ etex) ;
    zero.c = (0,0) ; % pourquoi pas ?

    numeric largeur ;
    largeur = xpart (un.c - zero.c) ;
    drawboxed(zero,un,deux,trois,quatre) ;

    % les huit points cardinaux des boîtes : n, ne, e, se, s, sw, w, nw
    fill (quatre.ne -- quatre.se -- (quatre.se shifted (4bp,0)) -- (quatre.ne shifted (4bp,0)) -- cycle)
            shifted (-2bp,0) withcolor white ;
    draw ((0,0) -- (largeur,0)) shifted trois.ne ;
    draw ((0,0) -- (largeur,0)) shifted trois.se ;
    draw ((largeur,0) -- (largeur + 2cm,0)) shifted trois.ne dashed evenly ;
    draw ((largeur,0) -- (largeur + 2cm,0)) shifted trois.se dashed evenly ;
    label.urt("0",zero.nw) ; label.urt("1",un.nw) ; label.urt(btex $n$ etex,trois.nw) ;
    label.top(btex \rlap{Table de hachage} etex,zero.nw shifted (0,16pt)) ;
    dotlabel("",trois.c) ;

    boxjoin(a.se = b.sw ; a.ne = b.nw) ; boxit.b0(btex $(x_0,y_0)$ etex) ; boxit.p0(" ") ;
    boxjoin(a.se = b.sw ; a.ne = b.nw) ; boxit.b1(btex $(x_1,y_1)$ etex) ; boxit.p1(" ") ;
    boxjoin(a.se = b.sw ; a.ne = b.nw) ; boxit.b2(btex $(x_2,y_2)$ etex) ; boxit.p2(" ") ;

    pair delta ;
    delta = (23mm,0) ;
    b0.c = trois.c shifted (-5mm,-35mm) ;
    b1.c = b0.c shifted delta ;
    b1.c = 1/2[b0.c,b2.c] ;
    drawboxed(b0,p0,b1,p1,b2,p2) ;

    boxjoin(a.se = b.sw ; a.ne = b.nw) ; boxit.bp(btex $(x_p,y_p)$ etex) ; boxit.pp(" ") ;
    bp.c = b2.c shifted (b2.c - b0.c) ;
    drawboxed(bp,pp) ;

    drawarrow trois.c{down}..{down}b0.c cutafter bpath b0 ;
    dotlabel("",p0.c) ; dotlabel("",p1.c) ; dotlabel("",p2.c) ; dotlabel("",pp.c) ;
    drawarrow p0.c{up}..{down}b1.c cutafter bpath b1 ;
    drawarrow p1.c{up}..{down}b2.c cutafter bpath b2 ;
    drawarrow p2.c{up}..{down}(b2.c shifted delta) cutafter (bpath b2) shifted delta ;
    drawarrow (1/2[p2.c,pp.c]){up}..{down}bp.c cutbefore (bpath p2) shifted delta cutafter bpath bp ;
    drawarrow pp.c{up}..{down}(bp.c shifted delta) ;
    % la terre
    z0 = bp.c shifted delta ;
    draw z0 - (4mm,0) -- z0 + (4mm,0) ;

    picture etape ; etape = currentpicture ; % garde le dessin courant
    currentpicture := nullpicture ; % on efface tout et on recommence
    path pochoir ; pochoir = z0 - (4mm,0) -- z0 + (4mm,0) -- z0 + (4mm,-2mm) -- z0 - (4mm,2mm) -- cycle ;
    for i=0 upto 14: draw ((0,0)--(-4mm,-4mm)) shifted ((z0 - (4mm,0)) shifted (i*mm,0)) ; endfor
    clip currentpicture to pochoir ;
    addto currentpicture also etape ;
endfig ;

end
```

3.7 Arbres binomiaux



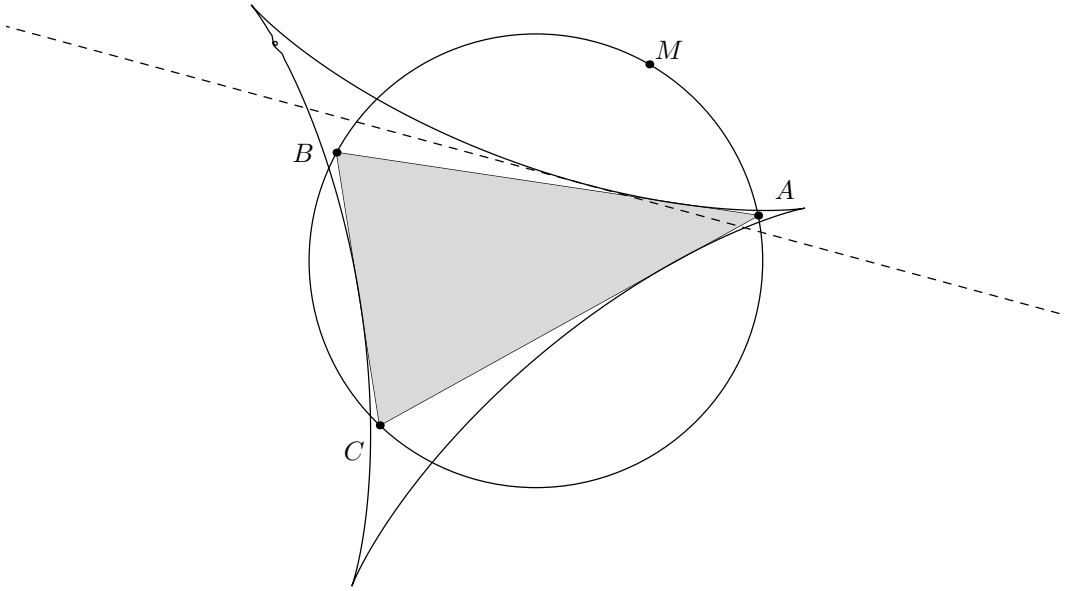
```
prologues := 2 ;
defaultfont := "CMR10" ;

beginfig(1)

    path t ;

    t = (0,0) -- (-30,-100) -- (30,-100) -- cycle ;
    fill t withcolor 0.8 white ;
    draw t ;

    draw (0,0) withpen pencircle scaled 5pt ;

    label(btex $B_n$ etex,center t) ;

    picture triangle ;

    triangle = currentpicture ;

    draw triangle shifted (-50,-50) ;

    draw (0,0) -- (-50,-50) ;

endfig ;

beginfig(2)

    picture bino[] ;

    def width(expr p) =
        xpart (lrcorner p - llcorner p)
    enddef ;

    bino[1] = nullpicture ;

    addto bino[1] doublepath (0,0) withpen pencircle scaled 5pt ;
    addto bino[1] doublepath (0,0) -- (0,-50) ;
    addto bino[1] doublepath (0,-50) withpen pencircle scaled 5pt ;

    for i = 2 upto 6 :
        bino[i] = bino[i - 1] ;
        addto bino[i] also (bino[i - 1] shifted (-width(bino[i-1]),0) shifted (-10,-50)) ;
        addto bino[i] doublepath (0,0) -- ((- 10,-50) shifted (-width(bino[i-1]),0)) ;
    endfor

    draw bino[4] rotated -90 ;
    draw (bino[6] shifted (width(bino[6]),0) shifted (20,0)) rotated -90 ;

endfig ;

end
```

3.8 Les cercles de Ford



```
prologues := 2 ;
defaultfont := "CMR10" ;

beginfig(1)

    def iteration(expr a,b,ra,rb,n) =
        if n = 0 :
        else :
            begingroup
                save t,m,r ;
                numeric t,r ;
                pair m ;
                t := sqrt(ra) + sqrt(rb) ;
                r := ra * rb / (t * t) ;
                m := (sqrt(ra)/t)[a,b] ;
                draw fullcircle scaled (r * 2u) shifted (m + (0,r*u)) ;
                iteration(a,m,ra,r,n-1) ;
                iteration(m,b,r,rb,n-1) ;
            endgroup
        fi
    enddef ;

    u = 10cm ;
    z0 = (0,0) ; z1 = (u,0) ;

    drawarrow (0,0)--(0,6cm) ; label.rt("y",(0,6cm)) ;
    drawarrow (0,0)--(11cm,0) ; label.bot("x",(11cm,0)) ;
    dotlabel.bot("1",z1) ; dotlabel.lrt("0",z0) ;

    draw fullcircle scaled u shifted (0,u/2) ;
    draw fullcircle scaled u shifted (u,u/2) ;

    pickup pencircle scaled 0.1bp ;

    iteration(z0,z1,1/2,1/2,6) ;

    clip currentpicture to (-1cm,-1cm)--(12cm,-1cm)--(12cm,7cm)--(-1cm,7cm)--cycle ;

endfig ;

end
```

3.9 Droites de Simpson et hypocycloïde de Steiner



```
prologues := 2 ;
defaultfont := "CMMI10" ;

beginfig(1)
    path cercle ;
    path triangle ;

    diametre = 6 cm ; rayon = diametre / 2 ;
    cercle = fullcircle scaled diametre ;
    draw cercle ;
% les sommets du triangle
    z1 = (rayon,0) rotated 11.5 ;
    z2 = z1 rotated 140 ;
    z3 = z2 rotated 75  ;
% le triangle
    triangle = z1 -- z2 -- z3 -- cycle ;
    draw triangle ;
    fill triangle withcolor 0.85 white ;
    labeloffset := 9bp ;
    dotlabel.urt("A",z1) ;
    dotlabel.lft("B",z2) ;
    dotlabel.llft("C",z3) ;
    labeloffset := 3bp ;
% la macro de projection orthogonale de a sur (bc)
    def projete(expr a,b,c) =
        begingroup
            save $ ;
            pair $ ;
            $ = whatever[b,c] ;
            ($ - a) rotated 90 shifted b = whatever[b,c] ;
            $
        endgroup
    enddef ;
% l'hypocycloïde de Steiner
    longueur = length cercle ;
    pair m[] ;
    pair p,q ;
    N = 250 ; % nombre de points sur le cercle
    for i = 1 upto N :
        p := projete(point (longueur*i/N) of cercle,z1,z2) ;
        q := projete(point (longueur*i/N) of cercle,z2,z3) ;
        m[i] = whatever[p,q] ;
        m[i-1] = whatever[p,q] ;
    endfor ;
    p := projete(point 0 of cercle,z1,z2) ;
    q := projete(point 0 of cercle,z2,z3) ;
    m[0] = whatever[p,q] ;

    draw m[0]
        for i = 1 upto N - 1 :
            .. m[i]
        endfor
        .. cycle ;
% tracé d'une droite de Simpson particulière
    z0 = (rayon,0) rotated 60 ;
    dotlabel.urt("M",z0) ;
    z4 = projete(z0,z1,z2) ;
    z5 = projete(z0,z2,z3) ;
    draw (-5)[z4,z5]--5[z4,z5] dashed evenly ;

% pour éviter tout débordement de la page
    clip currentpicture to (-7cm,-7cm)--(7cm,-7cm)--(7cm,7cm)--(-7cm,7cm)--cycle ;
endfig ;

end
```

4 Références

On trouvera toute la documentation utile, et tout particulièrement le manuel de référence de MetaPost par John Hobby, à l'adresse suivante :

`http://cm.bell-labs.com/who/hobby/mppubs.html`

On pourra également lire avec intérêt *The METAFONTbook*, de D. Knuth, chez Addison Wesley.