TIPE : Quantité d'encre utilisée par une fonte d'écriture

Listing du programme

Table des matières

1 Types

```
type commande =
  | Rmoveto
  | Vmoveto
  | Hmoveto
  | Rlineto
  | Vlineto
  | Hlineto
  | Rrcurveto
  | Vhcurveto
  | Hvcurveto
  |Closepath
  | Endchar
;;
type arguments = float list ;;
type instruction = commande * arguments ;;
type definition = instruction list ;;
type glyphe = char * definition ;;
type fonte = glyphe list ;;
type point = float * float ;;
type boundingBox = point * point ;;
```

2 Fonctions utilitaires

```
let rec pow_fl fl = function

  (* float -> int -> float *)

  (* Fonction puissance, appliquée à un flottant
      Utilise l'exponentiation rapide *)

  |0 -> 1.
  |1 -> fl
  |n -> pow_fl fl (n mod 2) *. r *. r where r = pow_fl fl (n/2)
;;

let rec pow x = function

  (* int -> int -> int *)

  (* Fonction puissance, appliquée à un entier
      Utilise l'exponentiation rapide *)

  |0 -> 1
  |1 -> x
  |n -> pow x (n mod 2) * r * r where r = pow x (n/2)
;;

let abs_fl = function

  (* float -> float *)

  (* Renvoie la valeur absolue d'un flottant *)

  |x when x >= 0. -> x
  |x              -> -.x
;;
```

```
32 let rec somme_liste = function

34   (* float list -> float *)

36   (* Fais la somme de tous les flottants de la liste (analogue de apply) *)

38   |[]    -> 0.
39   |t::q -> t +. somme_liste q
40 ;;


41 let norme1 (x,y) =

43   (* point -> float *)

45   (* Renvoie la norme 1 d'un point du plan dont les coordonnées sont des flottants *)

47   max (abs_fl x) (abs_fl y)
48 ;;


49 let distance1 (x1,y1) (x2,y2) =

51   (* point -> point -> float *)

53   (* Renvoie la distance liée à la norme 1 entre deux points dont les coordonnées sont des
         flottants *)

55   norme1 (x1-.x2,y1-.y2)
56 ;;


57 let fst_4 (e,_,_,_) =

59   (* 'a * 'b * 'c * 'd -> 'a *)

61   (* Renvoie la première composante d'un quadruplet *)

63   e
64 ;;


65 let snd_4 (_,e,_,_) =

67   (* 'a * 'b * 'c * 'd -> 'b *)

69   (* Renvoie la deuxième composante d'un quadruplet *)

71   e
72 ;;


73 let trd_4 (_,_,e,_) =

75   (* 'a * 'b * 'c * 'd -> 'c *)

77   (* Renvoie la troisième composante d'un quadruplet *)

79   e
80 ;;
```

```
81  let fth_4 (_,_,_,e) =

83    (* 'a * 'b * 'c * 'd -> 'd *)

85    (* Renvoie la quatrième composante d'un quadruplet *)

87    e
88  ;;


89  let rec int_of_float' x =

91    (* float -> int *)

93    (* Renvoie l'entier le plus proche (translation de la fonction partie entière) *)

95    if x >= 0.
96    then int_of_float (x+.0.5)
97    else - int_of_float' (-.x)
98  ;;


99  let float_of_int_2 (a,b) =

101   (* point -> point *)

103   (* Applique float_of_int à un point du plan *)

105   (float_of_int a,float_of_int b)
106 ;;


107 let int_of_float_2 (a,b) =

109   (* point -> int * int *)

111   (* Fonction int_of_float pour un couple *)

113   (int_of_float' a,int_of_float' b)
114 ;;


115 let somme_R2 (x1,y1) (x2,y2) =

117   (* point -> point -> point *)

119   (* Somme de deux points du plan *)

121   x1+.x2,y1+.y2
122 ;;


123 let prod_R2 k (x,y) =

125   (* float -> point -> point *)

127   (* Multiplication par un scalaire dans le plan *)

129   (k*.x,k*.y)
130 ;;
```

```
131 let diff_R2 a b =

133   (* point -> point -> point *)

135   (* Différence de deux points du plan *)

137   somme_R2 a (prod_R2 (-1.) b)
138 ;;
139 let init_graph () =

141   (* unit -> unit *)

143   (* Initialise la sortie graphique *)

145   moveto 0 0 ;
146   clear_graph() ;
147   set_color black
148 ;;
149 let rec applique f = function

151   (* ('a -> 'a -> 'a) -> 'a list -> 'a *)

153   (* Applique une fonction prenant deux arguments à une liste *)

155   |[e]   -> e
156   |t::q -> f t (applique f q)
157   |[]    -> failwith "cannot_apply_to_empty_list"
158 ;;
159 let min_list =

161   (* 'a list -> 'a *)

163   (* Renvoie le minimum d'une liste, avec une complexité linéaire *)

165   applique min
166 ;;
167 let max_list =

169   (* 'a list -> 'a *)

171   (* Renvoie le maximum d'une liste, avec une complexité linéaire *)

173   applique max
174 ;;
175 let list_of_string mot =

177   (* string -> char list *)

179   (* Décompose une chaîne de caractères en la liste de ses caractères *)


182   let res = ref [] in
183   for i = string_length mot - 1 downto 0 do
184     res := mot.[i] :: !res
185   done;
186   !res
187 ;;
```

```
188 let rec bar = function

190   (* (float * float) list -> float *)

192   (* Renvoie le barycentre des réels dont le poids est la deuxième composante du couple
193       Suppose que la somme des poids est égale à 1 *)

195   |[]       -> 0.
196   |(p,t)::q -> p *. t +. bar q
197 ;;



198 let bar2 l =

200   (* (point * float) list -> point *)

202   (* Renvoie le barycentre des points du plan dont le poids est la deuxième composante du
        couple
203       Suppose que la somme des poids est égale à 1 *)

205   bar (map (fun ((x,_),t) -> x,t) l),bar (map (fun ((_,y),t) -> y,t) l)
206 ;;



207 let attendre t =

209   (* float -> unit *)

211   (* Arrête le processus en cours pour une durée t, en secondes *)

213   let heureDebut= sys__time() in
214   while sys__time() < (heureDebut +.t) do
215     ()
216   done
217 ;;
```

3 Récupération d'une fonte

```
1 let lit fichier =

3   (* string -> string list *)

5   (* Renvoie la liste des lignes du fichier fichier *)

7   let entree = open_in fichier in
8   let rec creerListe acc =
9     try
10      creerListe (input_line entree :: acc)
11    with End_of_file -> rev acc
12  in
13  creerListe []
14 ;;
```

```
15 let split_ligne sep com ligne =

17   (* char -> char -> string -> string list *)

19   (* Sépare une chaîne de caractères en mots, délimités par sep, en s'arrêtant si le symbole
        de commentaire com est rencontré. Supprime par ailleurs les tabulations *)

21   let res = ref [] in
22   let mot = ref "" in
23   let stop = ref false in
24   let i = ref 0 in
25   while !i < string_length ligne && not !stop do
26     begin match ligne.[!i] with
27     |c when c=sep && !mot="" -> ()
28     |c when c=sep            -> res := !mot :: !res ; mot := ""
29     |c when c=com            -> stop := true
30     |c when c='\t'           -> ()
31     |c                       -> mot := !mot ^ string_of_char c
32     end ;
33     incr i
34   done;
35   if !mot = "" then rev !res
36   else rev (!mot :: !res)
37 ;;
```

```
38 let rec formate_ligne' acc = function

40   (* int list -> string list -> string * int list *)

42   (* Fonction auxiliaire de formate_liste, la généralisant *)

44   |[]    -> "",[]
45   |[a]   -> a,rev acc
46   |t::q -> formate_ligne' (int_of_string t::acc) q
47 ;;
```

```
48 let formate_ligne =

50   (* string list -> string * int list *)

52   (* Traite une ligne d'instruction de tracé PostScript, en renvoyant le couple de l'
        instruction, et de ses arguments, relatifs *)

54   formate_ligne' []
55 ;;
```

```
56 let recupereGlyphesEtSubrs fichier =

58   (* string list list -> (char * (string * int list) list) list * (int * (string * int list)
       list) list *)

60   (* Regroupe les caractères et les instructions les définissant dans une première liste, puis
       les sous-routines PostScript éventuelles et les instructions les définissant dans une
       seconde, à partir d'une liste d'instructions PostScript dont les lignes sont des listes
       de chaînes *)

62   let rec traite_ligne car n temp = function
63     |[]                                -> [],[]
64     |(t::p::q)::l when t = "dup"       -> traite_ligne ' ' (int_of_string p)
         [] l
65     |(t::p::q)::l when p = "NP"        -> glyphes,(n,rev temp)::subrs
66       where glyphes,subrs = traite_ligne ' ' 0 [] l
67     |("/space"::p::q)::l               -> traite_ligne ' ' 0 [] l
68     |("/exclam"::p::q)::l              -> traite_ligne '!' 0 [] l
69     |("/quotesingle"::p::q)::l         -> traite_ligne ''' 0 [] l
70     |("/parenleft"::p::q)::l           -> traite_ligne '(' 0 [] l
71     |("/parenright"::p::q)::l          -> traite_ligne ')' 0 [] l
72     |("/comma"::p::q)::l               -> traite_ligne ',' 0 [] l
73     |("/period"::p::q)::l              -> traite_ligne '.' 0 [] l
74     |("/colon"::p::q)::l               -> traite_ligne ':' 0 [] l
75     |("/semicolon"::p::q)::l           -> traite_ligne ';' 0 [] l
76     |("/question"::p::q)::l            -> traite_ligne '?' 0 [] l
77     |("/zero"::p::q)::l                -> traite_ligne '0' 0 [] l
78     |("/one"::p::q)::l                 -> traite_ligne '1' 1 [] l
79     |("/two"::p::q)::l                 -> traite_ligne '2' 2 [] l
80     |("/three"::p::q)::l               -> traite_ligne '3' 3 [] l
81     |("/four"::p::q)::l                -> traite_ligne '4' 4 [] l
82     |("/five"::p::q)::l                -> traite_ligne '5' 5 [] l
83     |("/six"::p::q)::l                 -> traite_ligne '6' 6 [] l
84     |("/seven"::p::q)::l               -> traite_ligne '7' 7 [] l
85     |("/eight"::p::q)::l               -> traite_ligne '8' 8 [] l
86     |("/nine"::p::q)::l                -> traite_ligne '9' 9 [] l
87     |(t::p::q)::l when t.[0] = '/'     -> traite_ligne t.[1] 0 [] l
88     |(t::p::q)::l when p = "ND"        -> (car,rev temp)::glyphes,subrs
89       where glyphes,subrs = traite_ligne ' ' 0 [] l
90     |(t::q)::l    when t = "return"   -> traite_ligne car n temp l
91     |t::l                              -> traite_ligne car n (formate_ligne t
         :: temp) l
92   in
93   traite_ligne ' ' 0 [] fichier
94 ;;




95 let traiteSubr subrs (com,l) = match com with

97   (* ('a * (string * 'a list) list) list ->  string * 'a list -> (string * 'a list) list *)

99   (* Renvoie la liste comprenant l'instruction passée en argument, en la remplaçant par sa
       description si c'est une sous-routine *)

101  |"callsubr" -> assoc (hd l) subrs
102  |_          -> [com,l]
103 ;;
```

```
104 let remplaceSubrs glyphes subrs =

106   (* ('a * (string * 'b list) list) list -> ('b * (string * 'b list) list) list -> ('a * (
          string * 'b list) list) list *)

108   (* Remplace les appels à des sous-routines par leurs descriptions exactes dans la
          description d'un glyphe *)

110   map (fun (c,d) -> c,flat_map (traiteSubr subrs) d) glyphes
111 ;;


112 let instructionReconnuep i =

114   (* string list *)

116   (* Vrai lorsque l'instruction de tracé est reconnue *)

118   mem i ["rmoveto";"vmoveto";"hmoveto";"rlineto";"vlineto";"hlineto";"
          rrcurveto";"vhcurveto";"hvcurveto";"closepath";"endchar"]
119 ;;


120 let commande_of_string = function

122   (* string -> commande *)

124   (* Convertit une chaîne de caractères en une commande, de type commande *)

126   |"rmoveto"   -> Rmoveto
127   |"vmoveto"   -> Vmoveto
128   |"hmoveto"   -> Hmoveto
129   |"rlineto"   -> Rlineto
130   |"vlineto"   -> Vlineto
131   |"hlineto"   -> Hlineto
132   |"rrcurveto" -> Rrcurveto
133   |"vhcurveto" -> Vhcurveto
134   |"hvcurveto" -> Hvcurveto
135   |"closepath" -> Closepath
136   |"endchar"   -> Endchar
137   |_           -> failwith "instruction_de_tracé_non_reconnue"
138 ;;


139 let rec convertitDefinition = function

141   (* (string * int list) list -> definition *)

143   (* Convertit une définition vers le type definition en convertissant les chaînes de caractè
          res vers le type commande et les entiers en flottants *)

145   |[] -> []
146   |(inst,args)::q when instructionReconnuep inst ->
147     (commande_of_string inst,map float_of_int args) :: convertitDefinition
          q
148   |_          ::q                                 ->
149     convertitDefinition q
150 ;;
```

```
151 let convertitGlyphes =

153   (* (char * (string * int list) list) list -> fonte *)

155   (* Convertit toutes les définitions d'une fonte vers le type definition *)

157   map (fun (c,d) -> c,convertitDefinition d)
158 ;;


159 let recupereFonte nom =

161   (* string -> fonte *)

163   (* Renvoie la description de la fonte dont le nom est donné en argument *)

165   let lignes = lit ("../Fontes/" ^ nom ^ "/" ^ nom ^ ".asm") in
166   let lignes_sep = map (split_ligne ' ' '%') lignes in
167   let glyphes,subrs = recupereGlyphesEtSubrs lignes_sep in
168   let glyphes_final = remplaceSubrs glyphes subrs in
169   let fonte = convertitGlyphes glyphes_final in
170   fonte
171 ;;
```

4 Détermination d'une bounding box

```
1  let rec boundingBoxGrossiere (x0,y0) = function

3    (* point_fl -> instruction_fl -> boundingBox_fl *)

5    (* Renvoie la bounding box des points de contrôle d'une instruction de tracée donnée *)

7    |Rlineto,[dx;dy]                              ->
8      (min x0 (x0+.dx),min y0 (y0+.dy)),(max x0 (x0+.dx),max y0 (y0+.dy))
9    |Vlineto,[dy]                                 ->
10     boundingBoxGrossiere (x0,y0) (Rlineto,[0.;dy])
11   |Hlineto,[dx]                                 ->
12     boundingBoxGrossiere (x0,y0) (Rlineto,[dx;0.])
13   |Rrcurveto,[dx1;dy1;dx2;dy2;dx3;dy3] ->
14     (min_list [x0;x0+.dx1;x0+.dx1+.dx2;x0+.dx1+.dx2+.dx3],
15      min_list [y0;y0+.dy1;y0+.dy1+.dy2;y0+.dy1+.dy2+.dy3]),
16     (max_list [x0;x0+.dx1;x0+.dx1+.dx2;x0+.dx1+.dx2+.dx3],
17      max_list [y0;y0+.dy1;y0+.dy1+.dy2;y0+.dy1+.dy2+.dy3])
18   |_ -> failwith "cas_non_envisagé"
19 ;;


20 let incluseBoundingBoxp ((xmin1,ymin1),(xmax1,ymax1)) ((xmin2,ymin2),(xmax2
     ,ymax2)) =

22   (* boundingBox -> boundingBox -> bool *)

24   (* Vrai lorsque la première bounding box est incluse dans la seconde *)

26   xmin1 > xmin2 && ymin1 > ymin2 && xmax1 < xmax2 && ymax1 < ymax2
27 ;;
```

```
28 let mergeBoundingBoxes ((xmin1,ymin1),(xmax1,ymax1)) ((xmin2,ymin2),(xmax2,
       ymax2)) =

30   (* boundingBox -> boundingBox -> boundingBox *)

32   (* Fusionne deux bounding box dans une bounding box plus grande *)

34   (min xmin1 xmin2,min ymin1 ymin2),(max xmax1 xmax2,max ymax1 ymax2)
35 ;;
```

```
36 let rec boundingBoxBezier p000 p001 p011 p111 =

38   (* point_fl -> point_fl -> point_fl -> point_fl -> boundingBox_fl *)

40   (* Renvoie la bounding box d'une cubique de bézier.
41      Les extrema sont recherchés par dichotomie, en utilisant le principe de l'algorithme de
           De Casteljau *)

43   if arret (p000,p001,p011,p111) 0.5 1.
44   then ((min_list [x0;x1;x2;x3],
45          min_list [y0;y1;y2;y3]),(max_list [x0;x1;x2;x3],
46                                   max_list [y0;y1;y2;y3]) where (x0,y0),(x1,
                                         y1),(x2,y2),(x3,y3)=p000,p001,p011,p111
                                         )
47   else mergeBoundingBoxes (boundingBoxBezier p000 p00t p0tt pttt) (
         boundingBoxBezier pttt ptt1 pt11 p111)
48     where (p000,p00t,p0tt,pttt),(_,ptt1,pt11,p111) = scinder (p000,p001,
           p011,p111) 0.5
49 ;;
```

```
50 let rec boundingBoxExacte (x0,y0) = function

52   (* point_fl -> instruction_fl -> boundingBox_fl *)

54   (* Renvoie la bounding box de la courbe dont l'instruction de tracé est donnée en argument
         *)

56   (*|commande,arguments -> ((xmin,ymin),(xmax,ymax))*)
57   |Rrcurveto,[dx1;dy1;dx2;dy2;dx3;dy3] ->
58     boundingBoxBezier
59       (x0,y0)
60       (x0+.dx1,y0+.dy1)
61       (x0+.dx1+.dx2,y0+.dy1+.dy2)
62       (x0+.dx1+.dx2+.dx3,y0+.dy1+.dy2+.dy3)
63   |Vhcurveto,[dy1;dx2;dy2;dx3]          ->
64     boundingBoxExacte (x0,y0) (Rrcurveto,[0.;dy1;dx2;dy2;dx3;0.])
65   |Hvcurveto,[dx1;dx2;dy2;dy3]          ->
66     boundingBoxExacte (x0,y0) (Rrcurveto,[dx1;0.;dx2;dy2;0.;dy3])
67   |instruction                          ->
68     boundingBoxGrossiere (x0,y0) instruction
69 ;;
```

```
70  let rec boundingBox' orig act box = function

72    (* point_fl -> point_fl -> boundingBox_fl -> boundingBox_fl *)

74    (* Fonction auxiliaire de boundingBox *)

76    |[]                                    ->
77      box
78    |(Rmoveto,[x;y])                  ::q ->
79      boundingBox' nouv nouv box q where nouv = somme_R2 act (x,y)
80    |(Rlineto,[x;y])                  ::q ->
81      boundingBox' orig (somme_R2 act (x,y)) (mergeBoundingBoxes box (
          boundingBoxExacte act (Rlineto,[x;y]))) q
82    |(Vlineto,[y]   )                 ::q ->
83      boundingBox' orig act box ((Rlineto,[0.;y])::q)
84    |(Hlineto,[x]   )                 ::q ->
85      boundingBox' orig act box ((Rlineto,[x;0.])::q)
86    |(Rrcurveto,[x1;y1;x2;y2;x3;y3])::q ->
87      if incluseBoundingBoxp (boundingBoxGrossiere act (Rrcurveto,[x1;y1;x2;
          y2;x3;y3])) box
88      then (boundingBox' orig (x0+.x1+.x2+.x3,y0+.y1+.y2+.y3) box q)
89      else boundingBox' orig (x0+.x1+.x2+.x3,y0+.y1+.y2+.y3) (
          mergeBoundingBoxes box (boundingBoxExacte act (Rrcurveto,[x1;y1;x2;
          y2;x3;y3]))) q
90        where x0,y0 = act
91    |(Vhcurveto,[y1;x2;y2;x3]      )::q ->
92      boundingBox' orig act box ((Rrcurveto,[0.;y1;x2;y2;x3;0.])::q)
93    |(Hvcurveto,[x1;x2;y2;y3]      )::q ->
94      boundingBox' orig act box ((Rrcurveto,[x1;0.;x2;y2;0.;y3])::q)
95    |(Vmoveto,[y])                    ::q ->
96      boundingBox' orig act box ((Rmoveto,[0.;y])::q)
97    |(Hmoveto,[x])                    ::q ->
98      boundingBox' orig act box ((Rmoveto,[x;0.])::q)
99    |(Closepath,_)                    ::q ->
100     boundingBox' orig act (mergeBoundingBoxes box (boundingBoxExacte act (
          Rlineto,[x;y]))) q
101     where x,y = diff_R2 orig act
102   |(Endchar,_)                      ::q ->
103     boundingBox' orig act (mergeBoundingBoxes box (boundingBoxExacte act (
          Rlineto,[x;y]))) q
104     where x,y = diff_R2 orig act
105   |_                                    ->
106     failwith "instruction_de_trace_non_reconnue"
107 ;;



108 let initialiseBoundingBox = function

110   (* definition -> boundingBox *)

112   (* Renvoie une bounding box nulle comprise dans la bounding box du glyphe *)

114   |(Rmoveto,[x;y])::_ -> (x,y),(x,y)
115   |(Hmoveto,[x])::_   -> (x,0.),(x,0.)
116   |(Vmoveto,[y])::_   -> (0.,y),(0.,y)
117   |_                  -> failwith "définition_ne_commence_pas_par_un_dé
        placement_du_point_courant"
118 ;;
```

```
119 let boundingBox definition =

121   (* definition -> boundingBox *)

123   (* Renvoie la bounding box d'un glyphe *)

125   boundingBox' (0.,0.) (0.,0.) bb0 definition where bb0 =
          initialiseBoundingBox definition
126 ;;



127 let largeur def =

129   (* definition -> int *)

131   (* Renvoie la largeur d'un glyphe *)

133   xmax -. xmin where (xmin,_),(xmax,_) = boundingBox def
134 ;;



135 let hauteur def =

137   (* definition -> int *)

139   (* Renvoie la hauteur d'un glyphe *)

141   ymax -. ymin where (_,ymin),(_,ymax) = boundingBox def ;;



142 let hauteurX fonte =

144   (* fonte -> int *)

146   (* Renvoie la hauteur d'x pour une fonte *)

148   hauteur (assoc 'x' fonte)
149 ;;



150 let rec boundingBoxPasAPas' = function (* Pour export vers Ocaml *)
151   |[e]   -> [boundingBox [e]]
152   |t::q -> boundingBox (rev (t::q)) :: (boundingBoxPasAPas' q)
153   |_     -> failwith "bounding_box_incalculable"
154 ;;



155 let boundingBoxPasAPas definition = (* Pour export vers Ocaml *)
156   let bb = boundingBoxPasAPas' (rev definition) in
157   let posMin = fst (hd bb) in
158   map (fun (m,M) -> int_of_float_2 (diff_R2 m posMin),int_of_float_2 (
          diff_R2 M posMin)) bb
159 ;;
```

5 Algorithme de De Casteljau

```
let rec pointsBezier' t pas (x0,y0) (x1,y1) (x2,y2) (x3,y3) =

  (* float -> float -> point -> point -> point -> point -> point list *)

  (* Renvoie les points (couples d'entiers) de la courbe paramétrée exacte, pour le pas pas.
     Les couples sont les points de contrôle, et t une variable auxiliaire, initialisée à 0
       et allant jusqu'à 1 *)

  if t > 1. then [x3,y3] (* Cas d'arrêt : on renvoie le dernier point de contrôle *)
  else let u = 1. -. t in (* Sinon on utilise les équations paramétrées *)
       let x = x0 *. pow_fl u 3 +. 3. *. x1 *. t *. pow_fl u 2 +. 3. *. x2
           *. pow_fl t 2 *. u +. pow_fl t 3 *.x3
       and y =y0 *. pow_fl u 3 +. 3. *.y1 *. t *. pow_fl u 2 +. 3. *.y2 *.
           pow_fl t 2 *. u +. pow_fl t 3 *.y3 in
       (x,y) :: pointsBezier' (t+.pas) pas (x0,y0) (x1,y1) (x2,y2) (x3,y3)
;;

let pointsBezier =

  (* float -> point -> point -> point -> point -> point list *)

  (* Renvoie les points (couples d'entiers) de la courbe paramétrée exacte, pour le pas pas *)

  pointsBezier' 0.
;;

let arret (p000,p001,p011,p111) t e =

  (* point * point * point * point -> float -> float -> bool *)

  (* Condition d'arrêt pour l'algorithme de De Casteljau
     Vrai lorsque tous les points de contrôle sont à une distance environ inférieure à e les
       uns des autres *)

  let pttt = bar2 [(p000,pow_fl t 3);(p001,3. *. pow_fl t 2 *. (1. -. t));(
      p011,3. *. t *. pow_fl (1. -. t) 2);(p111,pow_fl (1. -. t) 3)] in
  distance1 p000 pttt < e && distance1 p001 pttt < e && distance1 p011 pttt
      < e && distance1 p111 pttt < e
;;

let scinder (p000,p001,p011,p111) t =

  (* point * point * point * point -> float -> (point * point * point * point) * (point *
      point * point * point) *)

  (* À partir d'un quadruplet de points de contrôle d'une courbe de Bézier,
     renvoie deux tels quadruplets dont les points calculés par l'algorithme de De Casteljau
       définissent deux sous-courbes *)

  let u = 1. -. t in
  let p00t,p0t1,pt11 = bar2 [(p000,t);(p001,u)],bar2 [(p001,t);(p011,u)],
      bar2 [(p011,t);(p111,u)] in
  let p0tt,ptt1 = bar2 [(p00t,t);(p0t1,u)],bar2 [(p0t1,t);(pt11,u)] in
  let pttt = bar2 [(p0tt,t);(ptt1,u)] in
  (p000,p00t,p0tt,pttt),(pttt,ptt1,pt11,p111)
;;
```

```
45 let rec casteljau' e  = function

47   (* float -> (point * point * point * point) list -> (point * point * point * point) list *)

49   (* Renvoie la liste de quadruplets des points de contrôle des sous-courbes calculées par l'
         algorithme de De Casteljau *)

51   |[]                          -> []
52   |t::q when arret t 0.5 e -> t :: casteljau' e q
53   |t::q                        -> casteljau' e (t1::t2::q) where t1,t2 =
         scinder t 0.5
54 ;;
```

```
55 let casteljau x =

57   (* point * point * point * point -> (point * point * point * point) list *)

59   (* Application partielle de casteljau' pour un pas de 1 et les quatre premiers points de
         contrôle *)

61   casteljau' 1. [x]
62 ;;
```

```
63 let pointsCasteljau p q r s =

65   (* point -> point -> point -> point -> point list *)

67   (* Sélectionne, parmi les points de contrôle fournis par la fonction casteljau, ceux pré
         sents sur la courbe *)

69   (fst_4 (hd l))::(map fth_4 l) where l = casteljau (p,q,r,s)
70 ;;
```

6 Tracés de courbes

```
1 let rec traceCourbe' = function

3   (* point list -> unit *)

5   (* Relie les points de la liste en argument *)

7   |[]       -> ()
8   |(x,y)::q -> lineto (int_of_float' x) (int_of_float' y) ; traceCourbe' q
9 ;;


10 let traceCourbe ((x0,y0)::l) =

12   (* point list -> unit *)

14   (* Relie les points de la liste en argument en se déplaçant préalablement au premier *)

16   moveto (int_of_float' x0) (int_of_float' y0) ; traceCourbe' ((x0,y0)::l)
17 ;;
```

```
18 let rmoveto x0 y0 x y =
20   (* int -> int -> int -> int -> unit *)
22   (* Implémentation de l'analogue PostScript *)
24   moveto (x0+x) (y0+y)
25 ;;
26 let hmoveto x0 y0 x =
28   (* int -> int -> int -> unit *)
30   (* Implémentation de l'analogue PostScript *)
32   rmoveto x0 y0 x 0
33 ;;
34 let vmoveto x0 y0 =
36   (* int -> int -> int -> unit *)
38   (* Implémentation de l'analogue PostScript *)
40   rmoveto x0 y0 0
41 ;;
42 let rlineto x0 y0 x y =
44   (* int -> int -> int -> int -> unit *)
46   (* Implémentation de l'analogue PostScript *)
48   lineto (x0+x) (y0+y)
49 ;;
50 let vlineto x0 y0 =
52   (* int -> int -> int -> unit *)
54   (* Implémentation de l'analogue PostScript *)
56   rlineto x0 y0 0
57 ;;
58 let hlineto x0 y0 x =
60   (* int -> int -> int  -> unit *)
62   (* Implémentation de l'analogue PostScript *)
64   rlineto x0 y0 x 0
65 ;;
66 let  moveto_fl x  y =
68   (* float -> float -> unit *)
70   (* Implémentation de l'analogue PostScript, pour les flottants *)
72   moveto (int_of_float' x) (int_of_float' y)
73 ;;
```

```
74  let rmoveto_fl x0 y0 x y =
76    (* float -> float -> float -> float -> unit *)
78    (* Implémentation de l'analogue PostScript, pour les flottants *)
80    moveto_fl (x0+.x) (y0+.y)
81  ;;
82  let hmoveto_fl x0 y0 x =
84    (* float -> float -> float -> unit *)
86    (* Implémentation de l'analogue PostScript, pour les flottants *)
88    rmoveto_fl x0 y0 x 0.
89  ;;
90  let vmoveto_fl x0 y0 =
92    (* float -> float -> float -> unit *)
94    (* Implémentation de l'analogue PostScript, pour les flottants *)
96    rmoveto_fl x0 y0 0.
97  ;;
98  let  lineto_fl x  y =
100   (* float -> float -> unit *)
102   (* Implémentation de l'analogue PostScript, pour les flottants *)
104   lineto (int_of_float' x) (int_of_float' y)
105 ;;
106 let rlineto_fl x0 y0 x y =
108   (* float -> float -> float -> float -> unit *)
110   (* Implémentation de l'analogue PostScript, pour les flottants *)
112   lineto_fl (x0+.x) (y0+.y)
113 ;;
114 let vlineto_fl x0 y0 =
116   (* float -> float -> float -> unit *)
118   (* Implémentation de l'analogue PostScript, pour les flottants *)
120   rlineto_fl x0 y0 0.
121 ;;
122 let hlineto_fl x0 y0 x =
124   (* float -> float -> float -> unit *)
126   (* Implémentation de l'analogue PostScript, pour les flottants *)
128   rlineto_fl x0 y0 x 0.
129 ;;
```

```
130 let curveto x0 y0 x1 y1 x2 y2 x3 y3 =
132   (* float -> float -> float -> float -> float -> float -> float -> float -> unit *)
134   (* Implémentation de l'analogue PostScript *)
136   traceCourbe (pointsCasteljau (x0,y0) (x1,y1) (x2,y2) (x3,y3))
137 ;;

138 let rcurveto x0 y0 x1 y1 x2 y2 x3 y3 =
140   (* float -> float -> float -> float -> float -> float -> float -> float -> unit *)
142   (* Implémentation de l'analogue PostScript *)
144   curveto x0 y0 (x0+.x1) (y0+.y1) (x0+.x2) (y0+.y2) (x0+.x3) (y0+.y3)
145 ;;

146 let rrcurveto x0 y0 dx1 dy1 dx2 dy2 dx3 dy3 =
148   (* float -> float -> float -> float -> float -> float -> float -> float -> unit *)
150   (* Implémentation de l'analogue PostScript *)
152   rcurveto x0 y0 dx1 dy1 (dx1+.dx2) (dy1+.dy2) (dx1+.dx2+.dx3) (dy1+.dy2+.
        dy3)
153 ;;

154 let curveto_B x0 y0 x1 y1 x2 y2 x3 y3 =
156   (* float -> float -> float -> float -> float -> float -> float -> float -> unit *)
158   (* Implémentation de l'analogue PostScript, par le calcul naïf *)
160   traceCourbe (pointsBezier 0.001 (x0,y0) (x1,y1) (x2,y2) (x3,y3))
161 ;;

162 let rcurveto_B x0 y0 x1 y1 x2 y2 x3 y3 =
164   (* float -> float -> float -> float -> float -> float -> float -> float -> unit *)
166   (* Implémentation de l'analogue PostScript, par le calcul naïf *)
168   curveto_B x0 y0 (x0+.x1) (y0+.y1) (x0+.x2) (y0+.y2) (x0+.x3) (y0+.y3)
169 ;;

170 let rrcurveto_B x0 y0 dx1 dy1 dx2 dy2 dx3 dy3 =
172   (* float -> float -> float -> float -> float -> float -> float -> float -> unit *)
174   (* Implémentation de l'analogue PostScript, par le calcul naïf *)
176   rcurveto_B x0 y0 dx1 dy1 (dx1+.dx2) (dy1+.dy2) (dx1+.dx2+.dx3) (dy1+.dy2
        +.dy3)
177 ;;
```

```
178 let rec traceGlyphe' orig act = function

180   (* point -> point -> definition -> unit *)

182   (* Fontion auxiliaire de traceGlyphe *)

184   |[]                                    -> ()
185   |(Rmoveto,[x;y])                  ::q -> rmoveto_fl x0 y0 x y ;
186     traceGlyphe' nouv nouv q where nouv = somme_R2 act (x,y) and x0,y0 =
          act
187   |(Rlineto,[x;y])                  ::q -> rlineto_fl x0 y0 x y ;
188     traceGlyphe' orig (somme_R2 act (x,y)) q where x0,y0 = act
189   |(Vlineto,[y]   )                 ::q -> vlineto_fl x0 y0 y ;
190     traceGlyphe' orig (somme_R2 act (0.,y)) q where x0,y0 = act
191   |(Hlineto,[x]   )                 ::q -> hlineto_fl x0 y0 x ;
192     traceGlyphe' orig (somme_R2 act (x,0.)) q where x0,y0 = act
193   |(Rrcurveto,[x1;y1;x2;y2;x3;y3])::q -> rrcurveto x0 y0 x1 y1 x2 y2 x3 y3
        ;
194     traceGlyphe' orig (x0+.x1+.x2+.x3,y0+.y1+.y2+.y3) q where x0,y0 = act
195   |(Vhcurveto,[y1;x2;y2;x3]       )::q -> rrcurveto x0 y0 0. y1 x2 y2 x3 0.
        ;
196     traceGlyphe' orig (x0+.x2+.x3    ,y0+.y1+.y2    ) q where x0,y0 = act
197   |(Hvcurveto,[x1;x2;y2;y3]       )::q -> rrcurveto x0 y0 x1 0. x2 y2 0. y3
        ;
198     traceGlyphe' orig (x0+.x1+.x2    ,y0+.y2+.y3    ) q where x0,y0 = act
199   |(Vmoveto,[y])                    ::q -> vmoveto_fl x0 y0 y ;
200     traceGlyphe' nouv nouv q where nouv = somme_R2 act (0.,y) and x0,y0 =
          act
201   |(Hmoveto,[x])                    ::q -> hmoveto_fl x0 y0 x ;
202     traceGlyphe' nouv nouv q where nouv = somme_R2 act (x,0.) and x0,y0 =
          act
203   |(Closepath,_)                    ::q -> lineto_fl x0 y0 ;
204     traceGlyphe' orig act q where x0,y0 = orig
205   |(Endchar,_)                      ::q -> lineto_fl x0 y0 ;
206     traceGlyphe' orig act q where x0,y0 = orig
207   |_                                    -> failwith "
        instruction_de_trace_non_reconnue"
208 ;;


209 let traceGlyphe x =

211   (* point -> definition -> unit *)

213   (* Trace un glyphes à un point x donné à partir de sa définition *)

215   traceGlyphe' x x
216 ;;


217 let homothetie k =

219   (* float -> definition -> definition *)

221   (* Applique l'homothétie de rapport k à la définition d'un glyphe *)

223   map (fun (i,l) -> i,map (fun x -> x *. k) l)
224 ;;
```

```
225 let rotation' angle (x,y) =

227   (* float -> point -> point *)

229   (* Applique la rotation d'angle angle autour du point (0,0) à (x,y) *)

231   cos angle *. x +. sin angle *. y, -. sin angle *.x +. cos angle *. y
232 ;;
```

```
233 let rec rotation a = function

235   (* float -> definition -> definition *)

237   (* Applique la rotation d'angle angle autour du point (0,0) à la définition d'un glyphe *)

239   |[]                                    -> []
240   |(Rmoveto,[x;y])                  ::q -> (Rmoveto,[x';y']) :: rotation a q
        where x',y' = rotation' a (x,y)
241   |(Rlineto,[x;y])                  ::q -> (Rlineto,[x';y']) :: rotation a q
        where x',y' = rotation' a (x,y)
242   |(Vlineto,[y]   )                  ::q -> (Rlineto,[x';y']) :: rotation a q
        where x',y' = rotation' a (0.,y)
243   |(Hlineto,[x]   )                  ::q -> (Rlineto,[x';y']) :: rotation a q
        where x',y' = rotation' a (x,0.)
244   |(Rrcurveto,[x1;y1;x2;y2;x3;y3])::q -> (Rrcurveto,[x1';y1';x2';y2';x3';y3
        ']) :: rotation a q
245     where x1',y1' = rotation' a (x1,y1) and x2',y2' = rotation' a (x2,y2
          ) and x3',y3' = rotation' a (x3,y3)
246   |(Vhcurveto,[y1;x2;y2;x3]       )::q -> rotation a ((Rrcurveto,[0.;y1;x2;
        y2;x3;0.])::q)
247   |(Hvcurveto,[x1;x2;y2;y3]       )::q -> rotation a ((Rrcurveto,[x1;0.;x2;
        y2;0.;y3])::q)
248   |(Vmoveto,[y]   )                  ::q -> (Rmoveto,[x';y']) :: rotation a q
        where x',y' = rotation' a (0.,y)
249   |(Hmoveto,[x]   )                  ::q -> (Rmoveto,[x';y']) :: rotation a q
        where x',y' = rotation' a (x,0.)
250   |t                                   ::q -> t :: rotation a q
251 ;;
```

```
252 let traceBoundingBox p ((xmin,ymin),(xmax,ymax)) =

254   (* boundingBox -> unit *)

256   (* Trace une bounding box *)

258   traceCourbe (map (fun q -> somme_R2 p q) [xmin,ymin;xmin,ymax;xmax,ymax;
        xmax,ymin;xmin,ymin])
259 ;;
```

```
260 let rec arrivee pos = function

262   (* point -> instruction -> point *)

264   (* Donne la position d'arrivée après l'application d'une instruction, connaissant la
          position initiale *)

266   |(Rlineto,[x;y])                    -> somme_R2 pos (x,y)
267   |(Vlineto,[y]   )                   -> arrivee pos (Rlineto,[0.;y])
268   |(Hlineto,[x]   )                   -> arrivee pos (Rlineto,[x;0.])
269   |(Rrcurveto,[x1;y1;x2;y2;x3;y3]) -> somme_R2 pos (x1+.x2+.x3,y1+.y2+.y3)
270   |(Vhcurveto,[y1;x2;y2;x3]       ) -> arrivee pos (Rrcurveto,[0.;y1;x2;y2;
          x3;0.])
271   |(Hvcurveto,[x1;x2;y2;y3]       ) -> arrivee pos (Rrcurveto,[x1;0.;x2;y2
          ;0.;y3])
272   |_                                 -> failwith "instruction_de_tracé
          _non_reconnue"
273 ;;

274 let rec traceBoundingBoxesIndividuelles' orig pos couleurs = function

276   (* point -> point -> color list -> definition -> unit *)

278   (* Fonction auxiliaire de traceBoundingBoxesIndividuelles *)

280   |[]                                  -> ()
281   |(Rmoveto,[x;y])                  ::q -> traceBoundingBoxesIndividuelles'
          nouv nouv couleurs q where nouv = (somme_R2 pos (x,y))
282   |(Hmoveto,[x])                    ::q -> traceBoundingBoxesIndividuelles'
          orig pos couleurs ((Rmoveto,[x;0.])::q)
283   |(Vmoveto,[y])                    ::q -> traceBoundingBoxesIndividuelles'
          orig pos couleurs ((Rmoveto,[0.;y])::q)
284   |(Closepath,_)                    ::q ->
285     set_color (hd couleurs) ;
286     traceBoundingBox pos (boundingBoxExacte (0.,0.) (Rlineto,[x;y])) ;
287     traceBoundingBoxesIndividuelles' orig pos ((tl couleurs) @ [hd couleurs
          ]) q
288       where x,y = diff_R2 orig pos
289   |(Endchar,_)                      ::q ->
290     set_color (hd couleurs) ;
291     traceBoundingBox pos (boundingBoxExacte (0.,0.) (Rlineto,[x;y])) ;
292     traceBoundingBoxesIndividuelles' orig pos ((tl couleurs) @ [hd couleurs
          ]) q
293       where x,y = diff_R2 orig pos
294   |instruction                      ::q ->
295     set_color (hd couleurs) ;
296     traceBoundingBox pos (boundingBoxExacte (0.,0.) instruction) ;
297     traceBoundingBoxesIndividuelles' orig (arrivee pos instruction) ((tl
          couleurs) @ [hd couleurs]) q
298 ;;

299 let traceBoundingBoxesIndividuelles p =

301   (* point -> definition -> unit *)

303   (* Trace les bounding boxes relatives à chacune des parties d'un glyphe *)

305   traceBoundingBoxesIndividuelles' p p [red;green;blue;yellow;cyan;magenta]
306 ;;
```

```
307 let rec traceBoundingBoxPasAPas' p couleurs = function

309   (* point -> color list -> definition -> unit *)

311   (* Fonction auxiliaire de traceBoundingBoxPasAPas *)

313   |[]    -> ()
314   |t::q ->
315     traceBoundingBoxPasAPas' p ((tl couleurs) @ [hd couleurs]) q ;
316     set_color (hd couleurs) ;
317     attendre 2. ;
318     traceBoundingBox p (boundingBox (rev (t::q)))
319 ;;


320 let traceBoundingBoxPasAPas p glyphe =

322   (* point -> definition -> unit *)

324   (* Trace les différentes bounding boxes résultant des fusions successives lors du calcul de
        la bounding box d'un glyphe, à partir du point p *)

326   traceBoundingBoxPasAPas' p [red;green;blue;yellow;cyan;magenta] (rev (snd
        glyphe))
327 ;;


328 let rec ecrireMot' fonte pos k a = function

330   (* definition -> point -> float -> float -> char list -> string *)

332   (* Fonction auxiliaire de ecrireMot *)

334   |[]      -> ""
335   |' '::q ->
336     let l = (largeur (assoc 'x' fonte)) *. k in
337     let def = homothetie k (rotation a ([Hmoveto,[l]])) in
338     traceGlyphe pos def ;
339     "␣" ^ ecrireMot' fonte (somme_R2 pos (cos a *. l,-. sin a *. l)) k a q
340   |car::q ->
341     let l = (largeur (assoc car fonte)) *. k *. 1.1 in
342     let def = homothetie k (rotation a (assoc car fonte)) in
343     traceGlyphe pos def ;
344     string_of_char car ^ ecrireMot' fonte (somme_R2 pos (cos a *. l,-. sin
        a *. l)) k a q
345 ;;


346 let ecrireMot fonte pos k a mot =

348   (* definition -> point -> float -> float -> string -> string *)

350   (* Écrit, à partir du point pos, le mot mot, avec la fonte fonte, en appliquant l'homothétie
        de rapport k et la rotation d'angle a *)

352   ecrireMot' fonte pos k a (list_of_string mot)
353 ;;
```

```
354 let traceQuadrillage () =

356   (* unit -> unit *)

358   (* Trace un quadrillage adapté à une résolution 1600x900 *)

360   set_color (rgb 200 200 200) ;
361   for i = 0 to 20 do
362     moveto (72*i) 0 ;
363     rlineto (72*i) 0 0 10000 ;
364     moveto 0 (72*i) ;
365     rlineto 0 (72*i) 10000 0
366   done;
367   set_color black
368 ;;
```

7 Calculs d'aire

```
1  let rec eqParam (x0,y0) (cmd,args) =

3    (* point -> instruction -> (float -> float) * (float -> float) * point) *)

5    (*Renvoie le couple d'équations paramétrées de la courbe correspondant à l'instruction
         PostScript, et le point d'arrivée ; (a0,b0) est le point de départ *)

7    match cmd,args with
8    |Rlineto,[x;y]                      ->
9      (fun t -> x0 +. x *. t),(fun t -> y0 +. y *. t),(x0+.x,y0+.y)
10   |Hlineto,[x]                        -> eqParam (x0,y0) (Rlineto,[x;0.])
11   |Vlineto,[y]                        -> eqParam (x0,y0) (Rlineto,[0.;y])
12   |Rrcurveto,[x1;y1;x2;y2;x3;y3] ->
13     let a0,b0 = x0,y0 in
14     let a1,b1 = a0+.x1,b0+.y1 in
15     let a2,b2 = a1+.x2,b1+.y2 in
16     let a3,b3 = a2+.x3,b2+.y3 in
17     (fun t -> a0*.pow_fl (1.-.t) 3+.3.*.a1*.t*.pow_fl (1.-.t) 2+.3.*.a2*.
           pow_fl t 2*.(1.-.t)+.a3*.pow_fl t 3),
18     (fun t -> b0*.pow_fl (1.-.t) 3+.3.*.b1*.t*.pow_fl (1.-.t) 2+.3.*.b2*.
           pow_fl t 2*.(1.-.t)+.b3*.pow_fl t 3),
19     (x0+.x1+.x2+.x3,y0+.y1+.y2+.y3)
20   |_                                  -> (fun _ -> 0.),(fun _ -> 0.),(x0,y0)
21 ;;


22 let rectanglesGauche f a b n =

24   (* (float -> float) -> float -> float -> int -> float *)

26   (* Intégration numérique par la méthode des rectangles à gauche *)

28   let h = (b -. a) /. float_of_int n in
29   let res = ref 0. in
30   for i = 0 to n-1 do
31     res := !res +. f (a +. float_of_int i *. h);
32   done;
33   h *. !res
34 ;;
```

```
35 let simpson f a b n =

37   (* (float -> float) -> float -> float -> int -> float *)

39   (* Intégration numérique par la méthode de Simpson *)

41   let h = (b -. a) /. float_of_int n in
42   let res = ref 0. in
43   for i = 1 to n-1 do
44     res := !res +. f (a +. float_of_int i *. h) +. 2. *. f (a +. (
         float_of_int i +. 0.5) *. h)
45   done;
46   res := !res +. 2. *. (f a +. f b);
47   h *. !res /. 3.
48 ;;
```

```
49 let derive f h t =

51   (* (float -> float) -> float -> float -> float *)

53   (* Dérivée numérique *)

55   (f (t +. h) -. f (t -. h)) /. 2. /. h
56 ;;
```

```
57 let aireCubique (x0,y0) (x1,y1) (x2,y2) (x3,y3) =

59   (* point -> point -> point -> point -> float *)

61   (* Renvoie l'aire sous une cubique de Bézier, en utilisant la méthode de Simpson *)

63   let n = 10 in
64   let h = 1. /. float_of_int n in
65   let x t =
66     let u = 1. -. t in
67     pow_fl u 3 *.x0 +. 3. *. t *. pow_fl u 2 *.x1 +. 3. *. pow_fl t 2 *. u
         *.x2 +. pow_fl t 3 *.x3
68   in
69   let y t =
70     let u = 1. -. t in
71     pow_fl u 3 *.y0 +. 3. *. t *. pow_fl u 2 *.y1 +. 3. *. pow_fl t 2 *. u
         *.y2 +. pow_fl t 3 *.y3
72   in
73   simpson (fun t -> (x t *. (derive y h t)) -. y t *. (derive x h t)) 0. 1.
       n /. 2.
74 ;;
```

```
75  let rec aireGlyphe' orig pos = function

77    (* point -> point -> definition -> float *)

79    (* Fonction auxiliaire de aireGlyphe *)

81    |[]                                    -> 0.
82    |(Rmoveto,[x;y])                  ::q -> aireGlyphe' nouv nouv q where nouv
          = somme_R2 pos (x,y)
83    |(Rlineto,[x;y])                  ::q -> (x0*.y-.y0*.x)/.2. +. aireGlyphe'
          orig (somme_R2 pos (x,y)) q where x0,y0 = pos
84    |(Vlineto,[y]   )                 ::q -> aireGlyphe' orig pos ((Rlineto
          ,[0.;y])::q)
85    |(Hlineto,[x]   )                 ::q -> aireGlyphe' orig pos ((Rlineto,[x
          ;0.])::q)
86    |(Rrcurveto,[x1;y1;x2;y2;x3;y3])::q ->
87      let a1,a2 = pos in
88      let b1,b2 = somme_R2 (a1,a2) (x1,y1) in
89      let c1,c2 = somme_R2 (b1,b2) (x2,y2) in
90      let d1,d2 = somme_R2 (c1,c2) (x3,y3) in
91      ((6.*.c1+.3.*.b1+.a1)*.d2
92       +.(-.6.*.c2-.3.*.b2-.a2)*.d1
93       +.(3.*.b1+.3.*.a1)*.c2
94       +.(-.3.*.b2-.3.*.a2)*.c1
95       +.6.*.a1*.b2-.6.*.a2*.b1)
96      /.20.
97      +. aireGlyphe' orig (d1,d2) q
98    |(Vhcurveto,[y1;x2;y2;x3]       )::q -> aireGlyphe' orig pos ((Rrcurveto
          ,[0.;y1;x2;y2;x3;0.])::q)
99    |(Hvcurveto,[x1;x2;y2;y3]       )::q -> aireGlyphe' orig pos ((Rrcurveto,[
          x1;0.;x2;y2;0.;y3])::q)
100   |(Vmoveto,[y]   )                 ::q -> aireGlyphe' orig pos ((Rmoveto
          ,[0.;y])::q)
101   |(Hmoveto,[x]   )                 ::q -> aireGlyphe' orig pos ((Rmoveto,[x
          ;0.])::q)
102   |(Closepath,_   )                 ::q -> (x0*.y-.y0*.x)/.2. +. aireGlyphe'
          orig pos q
103     where x0,y0 = orig and x,y = diff_R2 orig pos
104   |_                                ::q -> aireGlyphe' orig pos q
105 ;;
```

```
106 let aireGlyphe =

108   (* definition -> float *)

110   (* Calcule l'aire d'un glyphe d'après sa définition *)

112   aireGlyphe' (0.,0.) (0.,0.)
113 ;;
```

```
114 let airesPonderees fonte =

116   (* fonte -> float *)

118   (* Calcule l'aire pondérée moyenne d'un glyphe pour une fonte *)

120   let freq = frequences "La_Bête_Humaine" in
121   let aires = make_vect 72 0. in
122   for i = 0 to vect_length aires - 1 do
123     aires.(i) <- aireGlyphe (assoc (char_of_int' i) fonte)
124   done;
125   let res = make_vect 72 0. in
126   for i = 0 to vect_length aires - 1 do
127     res.(i) <- freq.(i) *. aires.(i)
128   done;
129   (somme_tableau_fl res) /. (pow_fl (hauteurX fonte) 2)
130 ;;
```

8 Fréquences d'apparition des caractères

```
1   let incremente tab i =

3   (* int vect -> int -> unit *)

5   (* Incrémente la case i de tab de 1 *)

7   tab.(i) <- tab.(i) + 1
8 ;;
```

```
9 let int_of_char' = function

11  (* char -> int *)

13  (* Personnalisation de int_of_char *)

15    | ' '                          -> 0
16    | '!'                          -> 1
17    | ','                          -> 2
18    | '.'                          -> 3
19    | '?'                          -> 4
20    | t when int_of_char t < 39    -> -1
21    | t when int_of_char t < 42    -> int_of_char t - 34 (* '() *)
22    | t when int_of_char t < 48    -> -1
23    | t when int_of_char t < 58    -> int_of_char t - 38 (* 0-9 *)
24    | t when int_of_char t < 60    -> int_of_char t - 50 (* :; *)
25    | t when int_of_char t < 65    -> -1
26    | t when int_of_char t < 91    -> int_of_char t - 45 (* A-Z *)
27    | t when int_of_char t < 97    -> -1
28    | t when int_of_char t < 123   -> int_of_char t - 51 (* a-z *)
29    | _                            -> -1
30 ;;
```

```
31 let char_of_int' = function

33   (* int -> char *)

35   (* Personnalisation de char_of_int *)

37   |0                          -> ' '
38   |1                          -> '!'
39   |2                          -> ','
40   |3                          -> '.'
41   |4                          -> '?'
42   |n when n < 8               -> char_of_int (n + 34) (* '() *)
43   |n when n < 10              -> char_of_int (n + 50) (* :; *)
44   |n when n < 20              -> char_of_int (n + 38) (* 0-9 *)
45   |n when n < 46              -> char_of_int (n + 45) (* A-Z *)
46   |n when n < 72              -> char_of_int (n + 51) (* a-z *)
47   |_                          -> failwith "pas_de_caractere_correspondant"
48 ;;


49 let rec ajoute tab = function

51   (* int vect -> char list -> unit *)

53   (* Compte, par effet de bord, les lettres d'une ligne *)

55   |[]                              -> ()
56   |t::q when int_of_char' t = -1 -> ()
57   |t::q                            ->
58     incremente tab (int_of_char' t) ;
59     ajoute tab q
60 ;;


61 let rec parcourir tab = function

63   (* int vect -> string list -> unit *)

65   (* Compte, par effet de bord, les lettres d'un fichier *)

67   |[]   -> ()
68   |t::q -> ajoute tab (list_of_string t) ; parcourir tab q
69 ;;


70 let somme_tableau tab =

72   (* int vect -> int *)

74   (* Somme les composantes d'un vecteur d'entiers *)

76   let somme = ref 0 in
77   for i = 1 to vect_length tab - 1 do
78     somme := !somme + tab.(i)
79   done;
80   !somme
81 ;;
```

```
82 let somme_tableau_fl tab =

84   (* float vect -> int *)

86   (* Somme les composantes d'un vecteur de flottants *)

88   let somme = ref 0. in
89   for i = 1 to vect_length tab - 1 do
90     somme := !somme +. tab.(i)
91   done;
92   !somme
93 ;;

94 let frequences fichier =

96   (* string -> float vect *)

98   (* Calcule la fréquence d'apparition des caractères étudiés dans fichier *)

100  let tab = make_vect 72 0 in
101  let res = make_vect 72 0. in
102  let lignes = lit ("../Documents/Livres/" ^ fichier ^ ".txt") in
103  parcourir tab lignes ;
104  let somme = float_of_int (somme_tableau tab) in
105  for i = 1 to vect_length tab - 1 do
106    res.(i) <- (float_of_int tab.(i)) /. somme
107  done;
108  res
109 ;;
```