

PLAIN\_EX

## EXTENSION OF METAPOST plain FORMAT (ver. 0.46)

## ROZSZERZENIE METAPOST-owego FORMATU plain (ver. 0.46)

if known *plain-ex-ver*: **expandafter** **endinput** **else**: *plain-ex-ver* := .45; **fi**  
**def** *killtext* **text** *t* = **enddef**; % *absent from older versions of plain.mf*

Knuthian tradition:

| Knuthowa tradycja:

*mm*# = 2.84528; *pt*# = 1; *dd*# = 1.07001; *bp*# = 1.00375; *cm*# = 28.45276; *pc*# = 12;  
*cc*# = 12.84010; *in*# = 72.27;

A patch for bugs in METAPOST  
—turningnumber— functionality

| Makra łatające wadliwe funkcjonowanie operacji  
—turningnumber—

**let** *original\_turningnumber* = *turningnumber*;  
**vardef** *straighten\_path*(**expr** *r*) =  
  **for** *k* = 0 **upto** *length r* - 1: **point** *k* **of** *r* **-- endfor**  
**if** *cycle r*: *cycle* **else**: **point** *infinity* **of** *r* **fi** **enddef**;  
**vardef** *emergency\_turningnumber* **primary** *r* =  
  *original\_turningnumber*(*straighten\_path*(*r*))  
**enddef**;  
**def** *use\_emergency\_turningnumber* =  
  **def** *turningnumber* = *emergency\_turningnumber* **enddef**;  
**enddef**;  
**def** *use\_original\_turningnumber* =  
  **def** *turningnumber* = *original\_turningnumber* **enddef**;  
**enddef**;

Loading a file optionally

| Opcjonalne czytanie pliku

**boolean** *maybeinput\_ok*; **string** *maybeidir*;  
**vardef** *maybeiname*(**text** *name*) =  
  **if** known *maybeidir*: *maybeidir* & **fi** **if** *string name*: *name* **else**: **str** *name* **fi**  
**enddef**;  
**def** *maybeinput* **text** *name* =  
  **if**(**readfrom** *maybeiname*(*name*)) = *EOF*:  
    **message** "PX:␣file␣" & *maybeiname*(*name*) & "␣cannot␣be␣read";  
    *maybeinput\_ok* := **false**;  
  **else**:  
    **closefrom** *maybeiname*(*name*); **scantokens**("input␣" & *maybeiname*(*name*));  
    *maybeinput\_ok* := **true**;  
  **fi**  
**enddef**;

A few colours more:

| Parę dodatkowych kolorów:

**color** *cyan*, *magenta*, *yellow*; *cyan* = (0, 1, 1); *magenta* = (1, 0, 1); *yellow* = (1, 1, 0);

A few functions more:

| Parę dodatkowych funkcji:

**vardef** *gen\_whatever*(**text** *type*) = *save?*; *type*; ? **enddef**; % 1-argument *func*.  
**vardef** *whatever\_pair* = *gen\_whatever*(**pair**) **enddef**; % 0-argument *function*  
**vardef** *tand* **primary** *a* = *sind*(*a*)/ *cosd*(*a*) **enddef**;  
**vardef** *cotd* **primary** *a* = *cosd*(*a*)/ *sind*(*a*) **enddef**;  
**vardef** *signum* **primary** *x* = **if** *x* > 0: 1 **elseif** *x* < 0: -1 **else**: 0 **fi** **enddef**;  
**primarydef** *w\_dotnorm.z* =  
  **begingroup**  
    **save** *w\**, *z\**, *lw\**, *lz\**; **pair** *w\**, *z\**;  
    *lw\** = *abs*(*w*); *w\** := *w* **if** *lw\** > 0: /*lw\** **fi**;  
    *lz\** = *abs*(*z*); *z\** := *z* **if** *lz\** > 0: /*lz\** **fi**;  
    (*xpart w\** \* *xpart z\** + *ypart w\** \* *ypart z\**)  
  **endgroup**

```

enddef;
%
let ori_decimal = decimal;
def decimal primary n =
(
  if path n:
    for i* = 0 upto length(n) - 1: if i* > 0: &"_"& fi
    decimal(point i* of n) & "_" & decimal(postcontrol i* of n) & "_" &
    decimal(precontrol i* + 1 of n) & "_" & decimal(point i* + 1 of n)
  endfor
  elseif color n: ori_decimal(redpart(n)) & "_" &
    ori_decimal(greenpart(n)) & "_" & ori_decimal(bluepart(n))
  elseif pair n: ori_decimal(xpart(n)) & "_" & ori_decimal(ypart(n))
  else: ori_decimal(n) fi
)
enddef;

```

The definition of **postdir** and **predir** given below is based on the following observation, being the consequence of the de l'Hôpital's rule: consider a Bézier segment  $a \dots$  controls  $b$  and  $c \dots d$ ; normally, the vector  $\vec{ab}$  determines the "post" direction at node  $a$ ; if  $b$  coincides with  $a$ , then the vector  $\vec{ac}$  determines the direction; if also  $c$  coincides with  $a$ , then the last resort is the vector  $\vec{ad}$ ; if even  $d$  coincides with  $a$ , the Bézier segment is degenerated, and can be removed (a similar argumentation can be provided for the "pre" direction at node  $d$ ).

Definicje makr **postdir** i **predir** wykorzystują następującą obserwację, będącą prostą konsekwencją reguły de l'Hôpitala: rozważmy segment  $a \dots$  controls  $b$  and  $c \dots d$ ; w normalnej sytuacji kierunku „post” w punkcie  $a$  jest określony przez wektor  $\vec{ab}$ ; jeżeli punkty  $a$  i  $b$  się pokrywają, kierunek „post” w punkcie  $a$  jest określony przez wektor  $\vec{ac}$ ; jeżeli także punkty  $a$  i  $c$  się pokrywają, ostatnią szansą na wyznaczenie kierunku „post” w punkcie  $a$  jest wektor  $\vec{ad}$ ; jeżeli wszystkie punkty  $a, b, c, i d$  się pokrywają, to oznacza, że segment jest zdegenerowany i może być w zasadzie usunięty (analogiczną argumentację można przedstawić dla kierunku „pre” w punkcie  $d$ ).

```

% Previous, insufficiently robust definitions:
% vardef predir expr t of p = (point t of p) - (precontrol t of p) enddef;
% vardef postdir expr t of p = (postcontrol t of p) - (point t of p) enddef;
% vardef udir expr t of p = unitvector(direction t of p) enddef;
% New, more general definitions:
vardef gendir expr t of p =
  predir t of p + postdir t of p % direction-compatible definition
enddef;
vardef predir expr t of p =
  save a*, b*, c*, d*, s*, t*; pair a*, b*, c*, d*; path s*; t* := t;
  if not cycle p: if t < 0: t* := 0; elseif t > length(p): t* := length(p); fi fi
  s* = subpath(ceiling t* - 1, t*) of p;
  a* = point 0 of s*;
  b* = postcontrol 0 of s*; % b* <> postcontrol t - 1 of p for t = 0
  c* = precontrol 1 of s*;
  d* = point 1 of s*;
  if d* <> c*: d* - c*
  elseif d* <> b*: d* - b*
  elseif d* <> a*: d* - a*
  else: (0, 0)
  fi
enddef;
vardef postdir expr t of p =
  save a*, b*, c*, d*, s*, t*; pair a*, b*, c*, d*; path s*; t* := t;
  if not cycle p: if t < 0: t* := 0; elseif t > length(p): t* := length(p); fi fi
  s* = subpath(t*, floor t* + 1) of p;
  a* = point 0 of s*;

```

```

    b* = postcontrol 0 of s*;
    c* = precontrol 1 of s*; % c* <> precontrol t + 1 of p for t = length p
    d* = point 1 of s*;
    if a* <> b*: b* - a*
    elseif a* <> c*: c* - a*
    elseif a* <> d*: d* - a*
    else: (0, 0)
  fi
enddef;

```

---

% Definitions related to ‘pre-’ and  
‘post-’:

---

% Definicje jakoś związane z ‘pre-’  
i ‘post-’:

---

```

vardef udir expr t of p = unitvector(gendir t of p) enddef;
vardef upredir expr t of p = unitvector(predir t of p) enddef;
vardef upostdir expr t of p = unitvector(postdir t of p) enddef;
vardef pos_subpath expr z of p =
  if not cycle p: subpath z of p else:
    if xpart(z) <= ypart(z): subpath z of p
    else: subpath(xpart(z), ypart(z) + length(p)) of p fi
  fi
enddef;
%
vardef posttension expr t of p =% ‘The METAFONTbook’, ex. 14.15
  save q*; path q*;
  q* = point t of p{direction t of p} .. {direction t + 1 of p} point t + 1 of p;
  length(postcontrol 0 of q* - point 0 of q*)/
  length(postcontrol t of p - point t of p)% doesn’t work for ‘straight lines’
enddef;
vardef pretension expr t of p =% ditto
  save q*; path q*;
  q* = point t - 1 of p{direction t - 1 of p} .. {direction t of p} point t of p;
  length(precontrol 1 of q* - point 1 of q*)/
  length(precontrol t of p - point t of p)% doesn’t work for ‘straight lines’
enddef;

```

---

The two macros below, *path\_eq* and *inside* macros, might have been primitives. The macro *path\_eq* is obvious; *a* inside *b* returns true if the bounding box of *a* is inside the bounding box of *b*, which may be misleading; think, for example of: *fullcircle* inside *unitsquare* shifted  $(-1/2, -1/2)$  scaled .9 rotated 45. For most curves occuring in fonts, however, one can safely infer that if *a* inside *b* holds, then *a* is inside *b*.

---

Poniższe dwa makra mogłyby być w zasadzie instrukcjami podstawowymi. Makro *path\_eq* jest oczywiste. Wynikiem operacji *a* inside *b* jest „true” jeżeli prostokąt ograniczający *a* leży wewnątrz prostokąta ograniczającego *b*, co może dawać wynik mylący, np. w wypadku *fullcircle* inside *unitsquare* shifted  $(-1/2, -1/2)$  scaled 9 rotated 45. Jednakże w większości sytuacji występujących fontach można śmiało zakładać, że z *a* inside *b* wynika, iż *a* leży wewnątrz *b*.

---

```

vardef path_eq(expr a, b) =
  save i*, l*, r*; boolean r*;
  r* := (length(a) = length(b)) and (cycle a = cycle b);
  if r*:
    i* := 0; l* := length(a) if cycle a: -1 fi;
    forever:
      r* := (point i* of a = point i* of b); exitif not r*;
      r* := (precontrol i* of a = precontrol i* of b); exitif not r*;
      r* := (postcontrol i* of a = postcontrol i* of b); exitif not r*;
      exitif incr i* > l*;
    endfor fi
  r*
enddef;

```

```
%
tertiarydef a inside b =
  if path a: % and path b
    (xpart llcorner b < xpart llcorner a)and
    (xpart urcorner b > xpart urcorner a)and
    (ypart llcorner b < ypart llcorner a)and
    (ypart urcorner b > ypart urcorner a)
  else: % numeric a and pair b
    (a >= xpart b) and (a <= ypart b)
  fi
enddef;
```

---

% Two convenient macros more:

---

| % Jeszcze dwa wygodne makra:

---

```
vardef x_time expr x of p =% obsolete
  xpart(p intersectiontimes ((x, -infinity) -- (x, infinity)))
enddef;
vardef y_time expr y of p =% obsolete
  ypart(p intersectiontimes ((-infinity, y) -- (infinity, y)))
enddef;
vardef xtime expr x of p =% preferable alias
  xpart(p intersectiontimes ((x, -infinity) -- (x, infinity)))
enddef;
vardef ytime expr y of p =% preferable alias
  ypart(p intersectiontimes ((-infinity, y) -- (infinity, y)))
enddef;
```

---

A few figures more:

---

| Parę dodatkowych figur:

---

```
vardef triangle =
  (0, -1/2) -- (0.866, 0) -- (0, 1/2) -- cycle% 1/2 sqrt(3) ≈ 0.866025 ...
enddef;
vardef vpolygon(expr n) =
  for i := 0 upto n - 1: (1/2 right rotated ((360/n) * (i + 1/2))) -- endfor cycle
enddef;
```

---

A method, entangled a bit and not particularly robust, of testing whether a parameter is a *string* expression or a *suffix*. (Remark: `is_suffix((a))` or `is_suffix(a + b)` returns **true**; `is_suffix(((a)))` causes METAPOST to report an error).

---

Nieco pokrętna i niezbyt ogólna metoda sprawdzania, czy parametr jest wyrażeniem typu *string* czy *sufiksem*. (Uwaga: `is_suffix((a))` bądź `is_suffix(a + b)` zwraca **true**, zaś `is_suffix(((a)))` powoduje, że METAPOST zgłasza błąd).

---

```
vardef is_suffix(text suffix_or_not_suffix) =
  save the_suffix*; string the_suffix*; is_suffix* suffix_or_not_suffix;
  the_suffix* <> ""
enddef;
def is_suffix* suffix $ = the_suffix* := str $; killtext enddef;
```

---

The macro `&&` is to be used instead of the `&` operator if the respective ends of paths coincide only approximately; using `..` instead would add unwanted tiny Bézier segments. The macro is somewhat “left-handed,” i.e., it does not consider the expression that follow the macro, therefore, it can be used before the ‘cycle’ command; if the argument *p* of the macro `amp_amp*` is a **pair**, it is just ignored which may be considered hardly intuitive.

---

Makra `&&` należy używać zamiast operatora `&` jeśli końce ścieżek nie pokrywają się idealnie; użycie operatora `..` zamiast `&` spowodowałoby dodanie do ścieżki zbędnych (małych) segmentów Béziera. Makro to jest nieco „leworęczne”, tzn. nie analizuje wyrażenia, które się po nim pojawi, dzięki czemu może być użyte przed operatorem ‘cycle’; argument *p* makra `amp_amp*` będący punktem (**pair**) jest pomijany, co można uznać za zachowanie mało intuicyjne.

---

```
def && = amp_amp*whatever enddef;
tertiarydef p amp_amp* q =
  if not pair p:
```

(**subpath**(0, length( $p$ ) - 1) **of**  $p$ ) .. controls(**postcontrol** length( $p$ ) - 1 **of**  $p$ )  
and (**precontrol** length( $p$ ) **of**  $p$ ) ..

**fi**

**enddef**;

A few postfix and infix path operators:

Parę postfiksowych i infiksowych operatorów  
ścieżkowych:

**primarydef**  $a$  mirr  $b = a$  reflectedabout(**origin**,  $b$ ) **enddef**;

%

**def** store\_prec\_obj = store\_prec\_obj\**whatever* **enddef**;

**primarydef**  $a$  store\_prec\_obj\*  $b =$  *hide*(**def** prec\_obj =  $a$  **enddef**) **enddef**;

%

**primarydef**  $a$  sub  $b =$

if path  $a$ : (**pos\_subpath**  $b$  **of**  $a$ ) **elseif** string  $a$ : (**substring**  $b$  **of**  $a$ ) **fi**

**enddef**;

%

**def** node = store\_prec\_obj node\* **enddef**;

**vardef** node\*@# **primary**  $a =$

if str @# = "x": xpart(**point**  $a$  **of** prec\_obj)

elseif str @# = "y": ypart(**point**  $a$  **of** prec\_obj)

elseif str @# = "": **point**  $a$  **of** prec\_obj

else:

errhelp "The\_operator\_‘node’\_works\_only\_with\_‘x’,\_‘y’\_or\_an\_empty\_suffixes.";

errmessage "PX:improper\_usage\_of\_‘node’";

**fi**

**enddef**;

%

**def** first suffix \$ =

if str \$ = "at": % moves the first point of a path to a specified location

store\_prec\_obj prec\_obj shifted -(**point** 0 **of** prec\_obj)shifted

else: node \$(0) **fi**

**enddef**;

**def** last suffix \$ =

if str \$ = "at": % moves the last point of a path to a specified location

store\_prec\_obj prec\_objshifted

- (**point** if cycle prec\_obj: 0 **else**: infinity **fi** **of** prec\_obj)shifted

else: node \$(if cycle prec\_obj: 0 **else**: infinity **fi**) **fi**

**enddef**;

%

% node-governed flipping:

**def** nflipped = nflipped\**whatever* **enddef**;

**primarydef**  $a$  nflipped\*  $b =$

if cycle  $a$ :  $a$

else: reverse( $a$  reflectedabout(**point** 0 **of**  $a$ , **point** infinity **of**  $a$ ))

**fi**

**enddef**;

%

**def** xflipped = xflipped\**whatever* **enddef**;

**primarydef**  $a$  xflipped\*  $b =$

reverse( $a$  reflectedabout

( $1/2$  [llcorner  $a$ , lrcorner  $a$ ],  $1/2$  [ulcorner  $a$ , urcorner  $a$ ]))

**enddef**;

%

**def** yflipped = yflipped\**whatever* **enddef**;

**primarydef**  $a$  yflipped\*  $b =$

reverse( $a$  reflectedabout

( $1/2$  [llcorner  $a$ , ulcorner  $a$ ],  $1/2$  [lrcorner  $a$ , urcorner  $a$ ]))

**enddef**;

```
%
% node-governed rotating (infix operator):
primarydef a nrotated b =
  if cycle a: a
  else: a rotatedaround(1/2 [point 0 of a, point infinity of a], b)
fi
enddef;
%
% center-governed rotating (infix operator):
primarydef a crotated b =
  a rotatedaround(1/2 [llcorner a, urcorner a], b)
enddef;
```

---

Neat macros excerpted from John D. Hobby's  
boxes.mp macro package:

---

Zgrabne makra zaczerpnięte z zestawu makr  
boxes.mp Johna D. Hobby'ego:

---

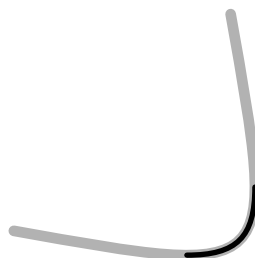
```
% Find the length of the prefix of string s for which cond is true for
% each character c of the prefix
vardef generisize_prefix(expr s)(text cond) =
  save i*, c*; string c*;
  i* = 0;
  forever:
    c* := substring(i*, i* + 1) of s;
    exitunless cond; exitif incr i* = length s;
  endfor
  i*
enddef;
%
% Take a string returned by the str operator and return the same string
% with explicit numeric subscripts replaced by generic subscript symbols []:
vardef generisize(expr s) =
  save res*, s*, l*; string res*, s*;
  res* = ""; % result so far
  s* = s; % left to process
  forever: exitif s* = "";
    l* := generisize_prefix(s*, (c* <> "[" and ((c* < "0") or (c* > "9"))));
    res* := res* & substring(0, l*) of s*;
    s* := substring(l*, infinity) of s*;
    if s* <> "":
      res* := res* & "[";
      l* := if s* >= "[": 1 + generisize_prefix(s*, c* <> "]")
      else: generisize_prefix(s*, (c* = ".") or ("0" <= c*) and (c* <= "9")) fi;
      s* := substring(l*, infinity) of s*;
    fi
  endfor
  res*
enddef;
```

The macro **extrapolate** computes a “superpath” (as opposed to “subpath”) for a single Bézier segment in such a way that the following identity holds (for  $0 \leq t_1 \leq t_2 \leq 1$ ):

$$\text{subpath}(t_1, t_2) \text{ of } (\text{extrapolate}(t_1, t_2) \text{ of } b) = b$$

Below, there are the results of the command **extrapolate**(.3, .7) of *p* for three similarly defined paths. The black line denotes the source path, the gray one—its extrapolation.

*p* = (0,0){right} .. {up}(s,s);



*p* = (0,0){right} .. tension .75 .. {up}(s,s);

*p* = (0,0){right} .. tension 3 .. {up}(s,s);



Exercise 1. What happens if the relation  $0 \leq t_1 \leq t_2 \leq 1$  is not fulfilled? (Hint: there are a few possible cases.)

Exercise 2. True or false:

**point 1 of**(**extrapolate**(*t<sub>a</sub>*, *t*) of *b*) = **point 1 of**(**extrapolate**(*t<sub>b</sub>*, *t*) of *b*) for  $t_a <> t_b$

Exercise 3. Try to imagine the result of the extrapolation for such weird (yet trivial) paths as:

(0, 0) .. controls(0, 0) and (100, 0) .. (100, 0) or (0, 0) .. controls(100, 0) and (0, 0) .. (100, 0)

Zadanie 1. Co by się stało, gdyby warunek  $0 \leq t_1 \leq t_2 \leq 1$  nie był spełniony? (Wskazówka: możliwych jest kilka różnych przypadków.)

Zadanie 2. Prawda czy fałsz:

Zadanie 3. Spróbuj przewidzieć wynik ekstrapolacji dla tak dziwnych (choć trywialnych) ścieżek jak:

**vardef** **extrapolate** **expr** *t* of *b* =% *t* pair, *b* Bézier segment

**clearxy**;

*Casteljau*(*xpart*(*t*)) = **point** 0 of *b*;

*Casteljau*( $\frac{1}{3}$  [*xpart*(*t*), *ypart*(*t*)] ) = **point**  $\frac{1}{3}$  of *b*;

*Casteljau*( $\frac{2}{3}$  [*xpart*(*t*), *ypart*(*t*)] ) = **point**  $\frac{2}{3}$  of *b*;

*Casteljau*(*ypart*(*t*)) = **point** 1 of *b*;

*z*<sub>0</sub> .. controls *z*<sub>1</sub> and *z*<sub>2</sub> .. *z*<sub>3</sub>

**enddef**;

%

**def** *Casteljau*(**expr** *t*) =

*t*[*t*[*t*[*z*<sub>0</sub>, *z*<sub>1</sub>], *t*[*z*<sub>1</sub>, *z*<sub>2</sub>]], *t*[*t*[*z*<sub>1</sub>, *z*<sub>2</sub>], *t*[*z*<sub>2</sub>, *z*<sub>3</sub>]]

**enddef**;

%

**vardef** *elongation\_to\_times*(**expr** *ea*, *eb*) =

% negative parameter values are admissible; they are meant for **pen\_stroke**

(if *ea* < 0: - *fl*/(abs(*ea*) + 1), *eb*/(abs(*eb*) + 1))

**enddef**;

A numerical function ‘*point\_line\_dist*’ takes as a parameter three **pair** expressions and returns a (signed) value of the distance of the first parameter from the line defined by the other two. It is referred to in the ‘*is\_line*’ function.

Funkcja ‘*point\_line\_dist*’, zależna od trzech wyrażeń typu **pair**, oblicza odległość (dodatnią lub ujemną) pierwszego z parametrów od linii określonej przez dwa pozostałe parametry. Do funkcji ‘*point\_line\_dist*’ odwołuje się funkcja ‘*is\_line*’.

**vardef** *point\_line\_dist*(**expr** *a*, *b*, *c*) =

**clearxy**; **save** *d*\*; *d*\* = sqrt(length(*b* - *c*));

*z*<sub>0</sub> = *a*/*d*\*; *z*<sub>1</sub> = *b*/*d*\*; *z*<sub>2</sub> = *c*/*d*\*;



$(x_2 - x_1) * (y_1 - y_0) - (x_1 - x_0) * (y_2 - y_1)$   
**enddef;**

The following code (its idea is due to Dan Luecking and Larry Siebenmann) computes the area surrounded by a cyclic path.

Poniższy kod (pomysł podsunęli Dan Luecking i Larry Siebenmann) oblicza pole obszaru ograniczonego zamkniętą krzywą Béziera.

```
newinternal area_scale;
area_scale := 1; % decrease if the result is going to be too large
vardef area(expr p) =% p is a Bézier segment; result =  $\int y dx$ 
  save xa, xb, xc, xd, ya, yb, yc, yd;
  (xa, 20ya) = point 0 of p;
  (xb, 20yb) = postcontrol 0 of p;
  (xc, 20yc) = precontrol 1 of p;
  (xd, 20yd) = point 1 of p;
  area_scale * (xb - xa) * (10ya + 6yb + 3yc + yd)
  + area_scale * (xc - xb) * (4ya + 6yb + 6yc + 4yd)
  + area_scale * (xd - xc) * (ya + 3yb + 6yc + 10yd)
enddef;
vardef Area(expr P) =% P is a cyclic path
  area(subpath(0, 1) of P)
  for t = 1 upto length(P) - 1: + area(subpath(t, t + 1) of P) endfor
enddef;
```

The idea of calculation of a turning angle between two vectors, employed in the definition of the function ‘turn\_ang,’ is based on the following observation:

Idea obliczania kąta (skierowanego) między dwoma wektorami, wykorzystana w funkcji ‘turn\_ang’, zasadza się na następującej obserwacji:

$$z \text{ reflectedabout}(\mathbf{origin}, \mathbf{right}) = 1/z$$

for a complex number  $z$  such that  $|z| = 1$ ; recall also that multiplication of complex numbers (‘zscaled’ operation) implies addition of their angle arguments.

dla liczby zespolonej  $z$  takiej, że  $|z| = 1$ ; przypomnijmy także, że mnożeniu liczb zespolonych (operacja ‘zscaled’) odpowiada dodawanie argumentów kątowych.

```
vardef turn_ang(expr za, zb) =
  if (abs(za) >= 1/1000) and (abs(zb) >= 1/1000): % eps may be not enough
    angle(unitvector(za) zscaled (unitvector(zb) reflectedabout(origin, right)))
  else: whatever fi
enddef;
```

A Boolean function ‘is\_line’ checks whether a given Bézier segment is a straight line. For large segments (fonts) it makes sense to specify a numerical parameter *is\_line\_off* >= 0; it defines a maximal acceptable distance of the control points of a Bézier arc from its secant (which corresponds to the distance between the arc and the secant circa  $3/4$  *is\_line\_off* for a symmetric, inflexionless arcs).

Boole’owska funkcja, ‘is\_line’ sprawdza, czy dany segment Béziera jest linią prostą. Dla dużych segmentów (fonty) może mieć sens zadanie parametru numerycznego *is\_line\_off* >= 0; określa on maksymalne dopuszczalne odchylenie naciągów krzywej Béziera od siecznej (co odpowiada odległości krzywej od siecznej wynoszącej ok.  $3/4$  *is\_line\_off* dla symetrycznych łuków bez punktów przegięcia).

```
vardef is_line(expr B) =
  save r*; boolean r*;
  if known is_line_off:
    save a*;
    a* = length((point 1 of B) - (point 0 of B));
    r* = (-a* + arclength(B)) <= (a*/infinity);
    if r*:
      r* := (is_line_off >= abs(point_line_dist(
        postcontrol 0 of B, point 0 of B, point 1 of B)))and
        (is_line_off >= abs(point_line_dist(
```

```

        precontrol 1 of B, point 0 of B, point 1 of B));
    fi
else: % backward compatibility
    save a*, b*, c*, d*;
    a* = length((point 1 of B) - (point 0 of B));
    b* = length((postcontrol 0 of B) - (point 0 of B));
    c* = length((precontrol 1 of B) - (postcontrol 0 of B));
    d* = length((point 1 of B) - (precontrol 1 of B));
    r* = (-a* + b* + c* + d* <= a*/infinity);
fi
r*
enddef;

```

---

Abbreviations for a few simple yet useful phrases: | Skróty dla kilku prostych acz przydatnych fraz:

---

```

def xyscaled primary p = xscaled xpart(p) yscaled ypart(p) enddef;
def yxscaled primary p = yscaled xpart(p) xscaled ypart(p) enddef;
primarydef a xscaledto b =
    hide(lastscale := b/(xpart(urcorner(a)) - xpart(llcorner(a))))
    a xscaled lastscale
enddef;
primarydef a xyscaledto b =
    hide(lastscale := b/(xpart(urcorner(a)) - xpart(llcorner(a))))
    a scaled lastscale
enddef;
primarydef a yscaledto b =
    hide(lastscale := b/(ypart(urcorner(a)) - ypart(llcorner(a))))
    a yscaled lastscale
enddef;
primarydef a yxscaledto b =
    hide(lastscale := b/(ypart(urcorner(a)) - ypart(llcorner(a))))
    a scaled lastscale
enddef;
%
pair lastshift;
primarydef a llshiftedto b =
    hide(lastshift := - llcorner(a) + b)a shifted lastshift
enddef;
primarydef a lrshiftedto b =
    hide(lastshift := - lrcorner(a) + b)a shifted lastshift
enddef;
primarydef a urshiftedto b =
    hide(lastshift := - urcorner(a) + b)a shifted lastshift
enddef;
primarydef a ulshiftedto b =
    hide(lastshift := - ulcorner(a) + b)a shifted lastshift
enddef;
primarydef a ccshiftedto b =
    hide(lastshift := - center(a) + b)a shifted lastshift
enddef;

```

---

Joining two paths at their intersection point: | Łączenie dwóch ścieżek w punkcie ich przecięcia:

---

```

tertiarydef a intersection_join b =% like softjoin
    begingroup save t*;
    (t1*, t2*) = a intersectiontimes b; a sub (0, t1*) && b sub (t2*, infinity)
    endgroup
enddef;

```

Changing locally non-internal variables (sometimes we want to set locally not only numeric variables):

Lokalna zmiana wartości zwykłych zmiennych (czasami zachodzi potrzeba lokalnej zmiany nie tylko zmiennych numerycznych):

**def local suffix  $s =$**

**begingroup**

**save**  $local\_stack\_value^*$ ,  $local\_stack\_name^*$ ;

**if** pair  $s$ : **pair**  $local\_stack\_value^*$ ; **fi**

**if** path  $s$ : **path**  $local\_stack\_value^*$ ; **fi**

**if** picture  $s$ : **picture**  $local\_stack\_value^*$ ; **fi**

**if** string  $s$ : **string**  $local\_stack\_value^*$ ; **fi**

**if** color  $s$ : **color**  $local\_stack\_value^*$ ; **fi**

$local\_stack\_value^* = s$ ; **def**  $local\_stack\_name^* = s$  **enddef**;

$local^*$

**enddef**;

**def**  $local^*$  **expr**  $x = local\_stack\_name^* := x$  **enddef**;

**def** **endlocal** =  $local\_stack\_name^* := local\_stack\_value^*$ ; **endgroup**; **enddef**;

The following abbreviation is roughly equivalent to the Knuthian *of-the-way function*, namely, to the  $whatever[z_1, z_2]$  operator; observe, however, that this construction requires that both  $z_1$  and  $z_2$  should be known, while  $z_1$  the in the construction  $z_1 \uparrow z_2$  can be unknown.

Poniższy skrót jest z grubsza równoważny Knuthowej funkcji *w-pół-drogi*, tzn. operatorowi  $whatever[z_1, z_2]$ ; odnotujmy wszakże, że konstrukcja ta wymaga, by zarówno  $z_1$  jak i  $z_2$  były znane, natomiast konstrukcja  $z_1 \uparrow z_2$  pozwala, by  $z_1$  było nieznane.

**primarydef**  $a \uparrow b = a + whatever * b$  **enddef**;

The macro 'leg' computes the leg of a right-angled triangle, given a hypotenuse (vector, parameter  $c$ ) and a length of one leg (parameter  $b$ ).

Makro 'leg' oblicza przyprostokątną trójkąta prostokątnego przy założeniu, że znana jest przeciwprostokątna (wektor, parametr  $c$ ) i przyprostokątna (parametr  $b$ ).

**primarydef**  $c \text{ leg } b =$

**begingroup** **save**  $a^*$ ; **pair**  $a^*$ ;

$a^* + b / (\text{length}(c) + - + b) * (a^* \text{ rotated } -90) = c$ ; %  $(\text{length}(c) + - + b) = \text{length}(a^*)$

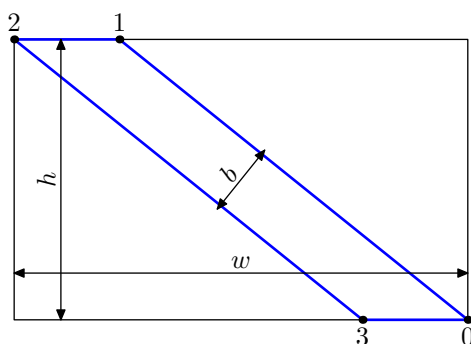
$a^*$

**endgroup**

**enddef**;

This extremely simple macro is particularly useful for constructing sloped objects. Assume, for example, that we want to draw the following parallelogram:

To prościutkie makro jest szczególnie użyteczne przy konstruowaniu ukośnych obiektów. Załóżmy, że chcemy narysować następujący równoległobok:



(given  $h$ ,  $w$  and  $b$ ). This is exactly the situation, where the macro 'leg' comes handy in. Given  $z_0$  and  $z_2$ , the remaining points  $z_1$  and  $z_3$  can be easily determined from the following relations:

dla zadanych wielkości  $h$ ,  $w$  i  $b$ . To jest właśnie ta sytuacja, w której makro 'leg' okazuje się użyteczne. Mając dane punkty  $z_0$  i  $z_2$ , punkty  $z_1$  i  $z_3$  wyznaczyć można z relacji:

$$z_1 = z_0 + whatever * ((z_2 - z_0) \text{ leg } (-b)); \quad y_1 = y_2; \quad z_1 - z_2 = z_0 - z_3;$$

Note the minus preceding the second argument to the ‘leg’; the positive value means “leftwards,” negative—“rightwards” (with respect to hypotenuse vector).

Odnotujmy obecność minusa przed drugim argumentem makra ‘leg’; wartość dodatnia oznacza „po lewej”, ujemna – „po prawej” (względem wektora przeciwprostokątnej).

The macro *quicksort* sorts  $@#s_i \dots @#s_j$  along with  $@#.\$i \dots @#.\$j$  for  $\$ \in t$ , using Tony Hoare’s “quick sort” method; suffix  $s$  must not occur in the  $t$  list (no checking is performed); if both  $s$  and  $t$  are empty,  $t$  is ignored.

- Remark 1: the algorithm has no explicit recursion, because of METAFONT/METAPOST limits on recursion level.
- Remark 2: the algorithm is not stable, i.e., it does not preserve the order of equal items.
- Sample usage: *quicksort*  $A(1, 100)()(x, y)$  will sort  $A_1, A_2, \dots, A_{100}$  (comparing  $A_i$  with  $A_j$ ) and, moreover,  $A_{x1}, A_{y1}, \dots, A_{x100}, A_{y100}$  will be reordered simultaneously.

Makro *quicksort* sortuje  $@#s_i \dots @#s_j$  wraz z  $@#.\$i \dots @#.\$j$  dla  $\$ \in t$ , z użyciem algorytmu “quick sort”, którego autorem jest Tony Hoare; sufiks  $s$  nie może wystąpić w liście  $t$  list (założenie to nie jest sprawdzane); jeśli parametry  $s$  i  $t$  są równocześnie puste, parametr  $t$  ignorowany.

- Uwaga 1: algorytm nie korzysta z jawnej rekursji ze względu na ograniczenia implementacyjne METAFONT/METAPOST.
- Uwaga 2: algorytm jest niestabilny, tzn. nie zachowuje kolejności równych obiektów.
- Przykładowe użycie: *quicksort*  $A(1, 100)()(x, y)$  posortuje  $A_1, A_2, \dots, A_{100}$  (porównując  $A_i$  z  $A_j$ ), ponadto  $A_{x1}, A_{y1}, \dots, A_{x100}, A_{y100}$  zostaną równocześnie odpowiednio przestawione.

```

vardef quicksort@#(expr i, j)(suffix s)(text t) =
  save i*, j*, k*, l*, cell*, stack*, incl_t*; boolean incl_t*;
  pair stack*[ ]; stack*_lev := 0; stack*[incr stack*_lev] := (i, j);
  i* := 0; for $ := t: i* := i* + 1; endfor% ‘‘measure’’ t-list
  incl_t* := (str s <> "") or ((str s = "") and (i* <> 0));
  forsuffixes $ := s if incl_t*: , t fi:
    if numeric @#$[i]: numeric cell$;
    elseif string @#$[i]: string cell$;
    elseif boolean @#$[i]: boolean cell$;
    fi
  endfor
  forever:
  exitif stack*_lev <= 0;
    numeric i*, j*; (i*, j*) = stack*[stack*_lev]; stack*_lev := stack*_lev - 1;
    if i* < j*:
      forsuffixes $ := s if incl_t*: , t fi: cell$ := @#$[i*]; endfor
      l* := i*;
      for k* := i* + 1 upto j*:
        if is_less(@#$[k*], cell$):
          forsuffixes $ := s if incl_t*: , t fi:
            @#$[l*] := @#$[k*]; @#$[k*] := @#$[l* + 1];
          endfor
          l* := l* + 1;
        fi
      endfor
      forsuffixes $ := s if incl_t*: , t fi: @#$[l*] := cell$; endfor
      stack*[incr stack*_lev] := (i*, l* - 1); stack*[incr stack*_lev] := (l* + 1, j*);
    fi
  endfor
enddef;
vardef is_less(expr a, b) = (a < b) enddef;

```

The macro *soften\_path* rounds all corners of a path *p*; *r* is the radius. Use *soften\_nodes* for rounding corners at a given set of nodes; its text parameter *t* is a comma-separated list of either numbers or pairs: a number means the number of a node, a pair means the number of a node and a local radius, to be used instead of *r*; prior to insertion the list of nodes is sorted.

Macro *soften\_path* zaokrągla naroża ścieżki *p*; *r* oznacza promień. Aby zaokrąglić naroża w wybranych węzłach należy użyć makra *soften\_nodes*; parametr tekstowy *t* tego makra jest listą (separowaną przecinkami) liczb lub par: liczba oznacza numer węzła, para – numer węzła i lokalny promień zaokrąglenia (zamiast *r*); lista węzłów przed wstawieniem jest sortowana.

```

vardef soften_node(expr p, r, t) =% path, radius, node (i.e., time)
  save q*; path q*; interim join_radius := r;
  if cycle p:
    q* = (subpath(t - 1, t) of p) softjoin (subpath(t, t + length(p) - 1) of p) & cycle;
    (subpath(1 - t, 1 + length(q*) - t) of q*) & cycle% re-position origin
  else: (subpath(0, t) of p) softjoin (subpath(t, length(p)) of p) fi
enddef;
%
vardef soften_nodes(expr p, r)(text t) =
  save j*, n*, p*, r*, t*; path p*; p* := p;
  t* := 0; for i* := t: (n*[incr t*], r*[t*]) = if pair i*: i* else: (i*, r) fi; endfor
  quicksort(1, t*)(n*)(r*);
  j* := -1; for i* := 1 upto t*: p* := soften_node(p*, r*[i*], n*[i*] + incr j*); endfor;
  p*
enddef;
%
vardef soften_path(expr p, r) =% path, radius
  save p*; path p*; p* := p;
  if r > 0:
    for i* := if cycle p: 0 else: 1 fi step 2 until 2(length(p) - 1):
      p* := soften_node(p*, r, i*);
    endfor;
  fi
  p*
enddef;

```

The macro *insert\_nodes* inserts additional nodes at given non-integer non-repeating times *t* into a given path *p*. The code would be a bit longer without ‘arclength’ and ‘arctime.’ The macro can be useful in some cases in the context of finding the envelopes of pen-stroked paths (avoiding inflection points—see below).

Macro *insert\_nodes* wstawia w ścieżce *p* dodatkowe węzły w punktach odpowiadających czasom *t* (niecałkowitym, niepowtarzającym się). Bez funkcji ‘arclength’ i ‘arctime’ kod byłby nieco dłuższy. Makro to może być przydatne przy wyznaczaniu obrysu piórka (unikanie punktów przegięcia – p. niżej).

```

vardef insert_nodes(expr p)(text t) =
  save j*, p*, s*, t*; path p*; p* := p;
  t* := 0;
  for i* := t:
    if round(i*) <> i*: % ignore integer times
      t*[incr t*] = arclength(subpath(0, i* mod length(p*)) of p*);
    fi
  endfor
  for i* := 1 upto t*:
    s* := arctime t*[i*] of p*;
    if abs(round(s*) - s*) > eps: % ignore repeating times; is eps OK?
      p* := (subpath(0, s*) of p*) && (subpath(s*, length p*) of p*)
      if cycle p*: & cycle fi;
    fi
  endfor;

```

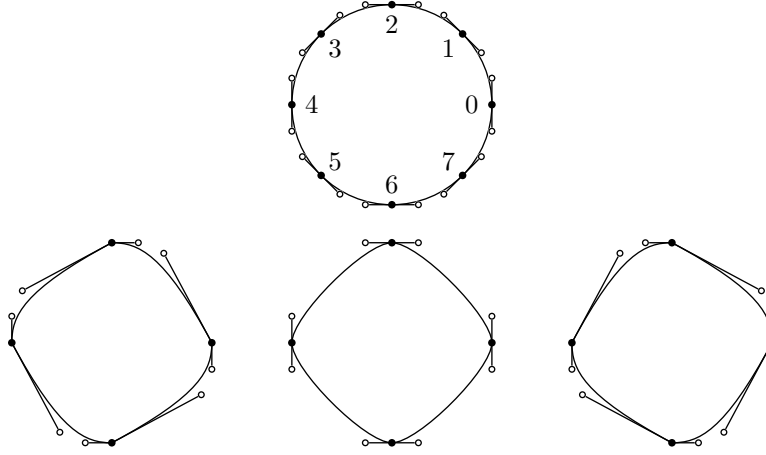
$p^*$   
**enddef;**

The macro *delete\_nodes* removes, as the name suggests, selected nodes from a given (cyclic) path. The macro can be useful for removing superfluous nodes, e.g., in case of improving fonts converted from PFB to METAPOST. The nodes can be removed in three possible ways: with both control nodes, with the precontrol node, or with the postcontrol one. During the process of removal, additional control nodes may be removed. More precisely: given a path  $p$ , let  $z^*[i^*]$  denotes its  $i^*$ -th node,  $z^*[i^*]b$ —the respective precontrol node (“before”),  $z^*[i^*]a$ —the respective postcontrol node (“after”). Macro *delete\_nodes* removes elements (by assigning an undefined value) from the tables  $z^*[\ ]$ ,  $z^*[\ ]b$  and  $z^*[\ ]a$  in such a way that the following invariant holds for the main loop: for any pair of consecutive known nodes  $z^*[i^*]$  and  $z^*[j^*]$ ,  $i^* < j^*$ , there exist exactly two indices  $u^*$  and  $v^*$  such that  $i^* \leq u^* < v^* \leq j^*$  and both  $z^*[u^*]a$  and  $z^*[v^*]b$  are known; moreover,  $z^*[w^*]a$  and  $z^*[w^*]b$  are unknown for the remaining indices  $w^*$ ,  $i^* \leq w^* \leq j^*$ . The resulting path is constructed from the elements that remain in the table.

Nodes to be removed are passed as the list of pairs (text parameter *node\_list*): (*index*, *kind*), where *index* denotes the time of the path corresponding to a given node, and *kind* is a number. If *kind* = 0, both control nodes are to be removed; if *kind* < 0, a precontrol node is to be removed; if *kind* > 0, a postcontrol node is to be removed.

The figure below shows the results of the *delete\_nodes* applied to a sample path  $p$  ( $p = \text{fullcircle}$ ): the topmost picture depicts the source path and the numbering of nodes, the left lower picture is the path returned by *delete\_nodes*( $p$ )((1, -1), (3, -1), (5, -1), (7, -1)) middle lower one—by *delete\_nodes*( $p$ )((1, 0), (3, 0), (5, 0), (7, 0)), right lower one—by *delete\_nodes*( $p$ )((1, 1), (3, 1), (5, 1), (7, 1)).

Makro *delete\_nodes*, jak sama nazwa wskazuje, usuwa wskazane węzły z danej (cyklicznej) ścieżki. Makro może się przydać do usuwania zbędnych węzłów, na przykład przy poprawianiu fontów zamienionych z postaci PFB na postać METAPOST-ową. Węzły są usuwane na trzy możliwe sposoby: wraz z obydwojoma przyległymi naciągami, z poprzedzającym lub z następującym naciągiem. W każdym z przypadków mogą zostać też usunięte sąsiadujące naciągi. Dokładniej: dla danej ścieżki  $p$  niech  $z^*[i^*]$  oznacza  $i^*$ -ty węzeł,  $z^*[i^*]b$  – jego naciąg poprzedzający,  $z^*[i^*]a$  – jego naciąg następujący. Makro *delete\_nodes* usuwa elementy (poprzez nadanie wartości nieokreślonej) z tablic  $z^*[\ ]$ ,  $z^*[\ ]b$  i  $z^*[\ ]a$  w taki sposób, by zachowany był następujący niezmiennik: dla dowolnej pary sąsiadujących węzłów  $z^*[i^*]$  i  $z^*[j^*]$ ,  $i^* < j^*$ , istnieją dokładnie dwa indeksy  $u^*$  i  $v^*$ , takie że  $i^* \leq u^* < v^* \leq j^*$  i zarówno  $z^*[u^*]a$ , jak i  $z^*[v^*]b$  są znane; ponadto  $z^*[w^*]a$  i  $z^*[w^*]b$  są nieokreślone dla pozostałych indeksów  $w^*$ ,  $i^* \leq w^* \leq j^*$ . Z elementów, które pozostają w tablicy, budowana jest ścieżka wynikowa. Węzły, które mają być usunięte, są podawane jako lista par (parametr tekstowy *node\_list*): (*indeks*, *rodzaj*), gdzie *indeks* oznacza czas odpowiadający danemu węzłowi na ścieżce, a *rodzaj* jest liczbą; jeśli *rodzaj* = 0, to usuwane są oba naciągi, jeśli *rodzaj* < 0, to usuwany jest naciąg poprzedzający, jeśli *rodzaj* > 0, to usuwany jest naciąg następujący. Efekt działania makra dla przykładowej ścieżki  $p$  ( $p = \text{fullcircle}$ ) jest przedstawiony na poniższym rysunku: górna ilustracja przedstawia ścieżkę źródłową z numeracją węzłów, lewa dolna ilustracja to rezultat użycia operacji *delete\_nodes*( $p$ )((1, -1), (3, -1), (5, -1), (7, -1)) środkowa dolna – operacji *delete\_nodes*( $p$ )((1, 0), (3, 0), (5, 0), (7, 0)), prawa dolna – operacji *delete\_nodes*( $p$ )((1, 1), (3, 1), (5, 1), (7, 1)).




---

```

vardef delete_nodes(expr p)(text node_list) =% p is cyclic
  save o*, i*, j*, j**, l*, n*, z*; pair z*[ ], z*[ ]a, z*[ ]b;
  n* = length(p);
  for i* := 0 upto n*:
    z*[i*] := point i* of p;
    z*[i*]a := postcontrol i* of p;
    z*[i*]b := precontrol i* of p;
  endfor
  for l* := node_list:
    % here the above-mentioned invariant holds (the result may depend
    % on the order of the nodes to be deleted, if neighbouring nodes
    % are being deleted)
    i* := xpart(l*) mod n*; % to mod or not to mod?
    % to ignore (with a message) or to apply the mod operation?
    z*[i*] := whatever_pair;
    if ypart(l*) = 0: % delete predeccessing b-node and successing a-node
      j* := i*; forever: exitif known z*[j*]b; j* := (j* - 1) mod n*; endfor
      z*[j*]b := whatever_pair;
      j* := i*; forever: exitif known z*[j*]a; j* := (j* + 1) mod n*; endfor
      z*[j*]a := whatever_pair;
    elseif ypart(l*) > 0: % delete successing a-node and b-node
      j* := i*; forever: exitif known z*[j*]a; j* := (j* + 1) mod n*; endfor
      z*[j*]a := whatever_pair;
      j* := i*; forever: j* := (j* + 1) mod n*; exitif known z*[j*]b; endfor
      z*[j*]b := whatever_pair;
    else: % ypart(l*) < 0; delete predeccessing a-node and b-node
      j* := i*; forever: j* := (j* - 1) mod n*; exitif known z*[j*]a; endfor
      z*[j*]a := whatever_pair;
      j* := i*; forever: exitif known z*[j*]b; j* := (j* - 1) mod n*; endfor
      z*[j*]b := whatever_pair;
    fi
  endfor
  i* := 0; forever: exitif known z*[i*]; i* := i* + 1; endfor; o* := i*;
  for i* := o* upto n* - 1 + o*:
    hide(j* := i* mod n*; j** := (i* + 1) mod n*)
    if known z*[j*]: z*[j*] fi
    if known z*[j*]a: .. controls z*[j*]a fi
    if known z*[j**]b: and z*[j**]b .. fi
  endfor z*[o*] & cycle
enddef;

```

The macros *insert\_extreme* and *insert\_extremes* insert additional nodes at extremes into a given path *p*. They make use of **unslant\_stroke** operation in order to provide valid extremes after slanting the path *p*. The first parameter (optional, of type **text**, i.e. it the list of numbers or pairs) defines either the minimal acceptable length of a resulting subpath (if its value is of the **numeric** type) or a subpath to be excluded from the process of the insertion of extremes (if its value is of the **pair** type).

Makra *insert\_extreme* i *insert\_extremes* wstawiają dodatkowe węzły w ekstremach w ścieżce *p*. Makra robi użytek z operacji **unslant\_stroke** aby zapewnić właściwe położenie ekstremów po pochyleniu ścieżki *p*. Pierwszy, opcjonalny parametr (typu **text**; tzn. jest to lista liczb lub par) pozwala podać długość minimalnej akceptowalnej (wynikowej) podścieżki oraz podścieżki nieuwzględniane przy wstawianiu ekstremów; decyduje o tym typ wartości parametru: *integer* oznacza długość, **pair** – podścieżkę.

```

vardef insert_extremes(text opt)(expr p)=% far from being optimal...
  save p*; path p*;
  p* := insert_extreme(opt)(up)(p);
  p* := insert_extreme(opt)(left)(p*);
  p* := insert_extreme(opt)(down)(p*);
  insert_extreme(opt)(right)(p*)
enddef;
%
vardef insert_extreme(text opt)(expr d, p) =
  save ignored_size*, ignored_segments*, a*, b*, found*, p*, t*, t**; path p*;
  ignored_size* := ignore_nib_limit;
  for opt* := opt:
    if numeric opt*: ignored_size* := opt*; fi
    if pair opt*:
      a* := floor(xpart opt*); b* := ceiling(ypart opt*) - 1; if b* < a*: b* := a*; fi
      for t* := a* upto b*: ignored_segments*[t*] := 1; endfor
    fi
  endfor
  for t* := 0 upto length(p) - 1:
    hide(found* := whatever)
    if unknown ignored_segments*[t*]:
      hide(p* := subpath(t*, t* + 1) of p)
      if arclength(p*) > ignored_size*:
        hide(t** := directiontime d unslant_stroke of p*)
        if (arclength(subpath(0, t**) of p*) > ignored_size*) and
          (arclength(subpath(t**, 1) of p*) > ignored_size*): hide(found* := 1)
        fi
      fi
    fi
    if known found*: ((subpath(0, t**) of p*) && (subpath(t**, 1) of p*)) else: p* fi
    &
  endfor
  if cycle p: cycle fi
enddef;

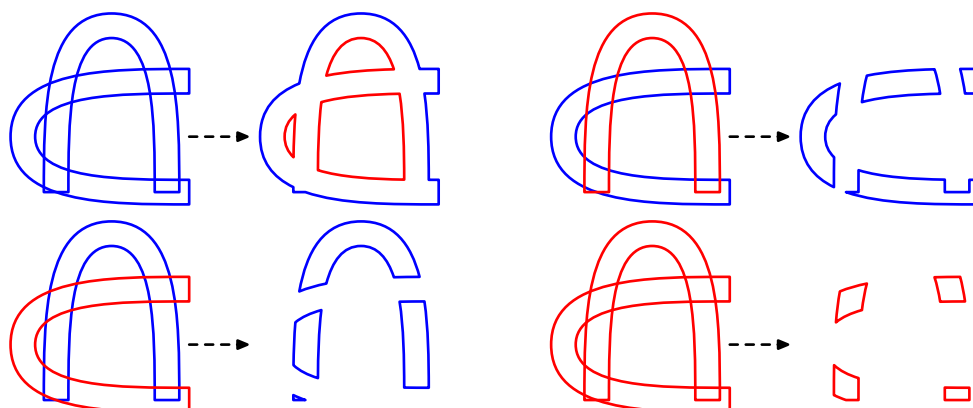
```



# A MODULE THAT FINDS AN OUTLINE FOR A SINGLE (SELF-INTERSECTING) CYCLIC PATH OR A PAIR OF CYCLIC PATHS

# MODUŁ ZNAJDUJĄCY OBWIEDNIĘ POJEDYNCZEJ ŚCIEŻKI CYKLICZNEJ (SAMOPRZECINAJĄCEJ SIĘ) LUB PARY ŚCIEŻEK CYKLICZNYCH

The problem can be stated as follows: two paths are given (precisely: expressions of type **path**); assume that the positively directed (anti-clockwise) path accomplishes filling, and negatively directed (clockwise)—erasing; the task is to find the outline of the resulting (visible) figure. Such a task is known as “removing overlaps” which seems too narrow for such a complex operation. Actually, the basic macro of that part, i.e., ‘find\_outlines,’ accomplishes set-theory operations: sum, difference and product, depending on the turning number of the input paths. The illustration below demonstrates the results yielded by the macro ‘find\_outlines.’ There are four cases since there are four combinations of turning numbers for two “regular” paths. Each case shows the initial situation (left) and the resulting one (right). Filling is omitted, the outline colour shows the turning number: blue—positive, red—negative.



A single (cyclic) curve can be supplied as an input argument to **find\_outlines**; in this case it *should have* self-intersections. Depending on the orientation of the input path, there are possible two sets of resulting paths:



Problem można postawić w następujący sposób: dane są dwie ścieżki (dokładniej wyrażenia typu **path**); zakładamy, że dodatnio skierowane ścieżki (przeciwnie do ruchu wskazówek zegara) zaczerniają obszar, a ujemnie – odczerniają. Należy znaleźć obwiednię tak powstałej (widocznej) figury. Operacja ta nosi nazwę „usuwanie części wspólnej”, co niezbyt dobrze oddaje złożony charakter operacji. W istocie podstawowa operacja tego modułu, tj. ‘find\_outlines’, realizuje operacje teoriomnogościowe: sumę, iloczyn i różnicę, w zależności od kierunków wejściowych ścieżek. Poniższa ilustracja przedstawia wyniki działania makra ‘find\_outlines’. Każdy z czterech przypadków pokazuje stan początkowy (lewa strona) i końcowy (prawa strona). Wypełnienie nie zostało naniesione, zaznaczony jedynie został kierunek ścieżek: kolor niebieski – kierunek dodatni, kolor czerwony – kierunek ujemny.

Argumentem wejściowym makra **find\_outlines** może być pojedyncza (cykliczna) ścieżka; w takim przypadku *powinna* ona mieć samoprzecięcia. Makro **find\_outlines** – w zależności od kierunku wejściowej ścieżki – znajduje dwa możliwe zestawy ścieżek wynikowych:

```
vardef feasible_cross(suffix p, q, v)(expr c, s) =
  if v[c]: (s * turn_ang(gendir p[c] of p, gendir q[c] of q) > 0)
  else: false fi
enddef;
%
```

```

vardef recombine(expr self_intersection)(suffix p, r) =
  save s*, e*, v*, n*; boolean v*[];
  if not path r0: scantokens("path_" & generisize(str r) & "[]"); fi
  if not numeric rnum: scantokens("numeric_" & generisize(str rnum); fi
  rnum := 0;
  s* := if self_intersection: 1 else: turningnumber(p1) * turningnumber(p2) fi;
  for i* := 1 upto pnum: v*[i*] := true; endfor
  forever:
    save c*; c* := 0;
    forever:
      c* := c* + 1; exitif c* > pnum; exitif feasible_cross(p1, p2, v*, c*, s*);
    endfor
    exitif c* > pnum;
    e* := 1;
    r[incr rnum] :=
      forever:
        hide(e* := 3 - e*; v*[c*] := false;
        if self_intersection: % identify the same crossings as ‘‘visited’’
          for i* := 1 upto pnum:
            % intersectiontime is not a comutative operation:
            if abs(p[e*][c*] - p[3 - e*][i*]) <= 16epsilon: % 1epsilon? 2epsilon?
              v*[i*] := false;
            fi
          endfor
        fi
        n* := next_time(p[e*])(c*)
        (pos_subpath(p[e*][c*], p[e*][n*]) of p[e*])hide(c* := n*)&&
        exitif not v*[n*];
      endfor
    cycle;
  endfor
enddef;
%
vardef next_time(suffix p)(expr t) =
  save c*;
  for i* := 1 upto pnum:
    if (p[i*] > p[t]):
      if known c*: if p[i*] < p[c*]: c* := i*; fi else: c* := i*; fi
    fi
  endfor
  if unknown c*:
    c* := 1; for i* := 2 upto pnum: if p[i*] < p[c*]: c* := i*; fi endfor
  fi
  c*
enddef;
%
vardef feasible_time(suffix p)(expr t) =
  save b*, i*; boolean b*; b* := true; i* := 0;
  forever:
    i* := i* + 1;
    exitif (unknown p[i*]) or (not b*);
    b* := b* if (abs(p[i*] - t) < 1) or (abs(p[i*] - t) > length(p) - 1): % optimization?
      and (arclength(pos_subpath(p[i*], t) of p) > acc_eps)
      and (arclength(pos_subpath(t, p[i*]) of p) > acc_eps)
    fi;
  endfor
  b*
enddef;

```

```

%
vardef intersect_curves(expr self_intersection)(suffix p) =
  save l*, p*; l* := 0;
  for i* := 0 upto length(p1) - 1:
    for j* := 0 upto length(p2) - 1:
% if a curve is being self-intersected, we demand that its neighbouring
% Bézier segments do not intersect (could be circumvent at the cost
% of the increased complexity of computation):
      if if self_intersection: (abs(i* - j*) mod (length(p1) - 1)) > 1 else: true fi:
        for k* := 1 upto
          intersect_segments(subpath(i*, i* + 1) of p1, subpath(j*, j* + 1) of p2)(p*):
% assuming that there are no self-intersections of single Bézier segments,
% it suffices to check feasibility only for p1:
          if feasible_time(p1, p1[k*] + i*):
            p1[incr l*] = p1[k*] + i*; p2[l*] = p2[k*] + j*;
          fi
        endfor
      fi
    endfor
  endfor
  pnum := p1num := p2num := l*;
enddef;
%
vardef intersect_segments(expr a, b)(suffix p) =
  save ta*, tb*; (ta*, tb*) = a intersectiontimes b;
  if ta* >= 0:
    p1[1] := ta*; p2[1] := tb*;
    save tc*, td*; (tc*, td*) = reverse a intersectiontimes reverse b;
    if length(1/2 [point ta* of a, point tb* of b]
      - 1/2 [point tc* of reverse a, point td* of reverse b]) > acc_eps:
      p1[2] := 1 - tc*; p2[2] := 1 - td*; 1+ fi 1
    else: 0 fi
enddef;
newinternal acc_eps; acc_eps := .5;

```

As was mentioned, the input argument to ‘find\_outlines’ can be either a single path or a pair of paths. The paths must comply with the following assumptions:

- paths are cyclic;
- if two paths are passed as an argument, each of them should contain no self-intersection;
- if a single path is passed as an argument, its adjacent segments do not intersect;
- no tangent touching occur;
- no inflection points occur (because the applied algorithm assumes that two Bézier segments can cross at at most two points)
- segments are long enough (minimal length of a segment after the process of intersection is controlled by the variable *acc\_eps*—see above).

Jak zostało wspomniane, wejściowym argumentem makra ,find\_outlines’ może być albo pojedyncza ścieżka, albo para ścieżek. Następujące warunki muszą być spełnione:

- ścieżki powinny być cykliczne
- jeśli argumentem jest para ścieżek, żadna z nich nie powinna mieć punktów samoprzecięcia;
- jeśli argumentem jest pojedyncza ścieżka, to jej sąsiednie segmenty Béziera nie powinny się przecinać;
- nie powinny występować punkty styczności;
- nie powinny występować punkty przegięcia (gdyż zastosowany algorytm zakłada, że dwa segmenty Béziera przecinają się co najwyżej w dwóch punktach);
- poszczególne segmenty powinny być dostatecznie długie (minimalna długość segmentu po procesie znajdowania przecięć jest określona przez zmienną *acc\_eps* – p. wyżej).

Further improvements:

- intersection points nearly coinciding with nodes should be replaced by the nodes, i.e., by an integer time.

Przewidywane udoskonalenie:

- jeśli punkt przecięcia niemalże pokrywa się z węzłem którejś z krzywych, to do obliczeń powinien być brany tenże węzeł.

```

vardef find_outlines(text a)(suffix r)=% a -- input, r -- output
  save auto*, i*, q*;
  boolean auto*; numeric qnum; numeric q*[ ][]; path q*[ ];
  i* := 0;
  for a* := a: q*[incr i*] := a*; endfor
  if i* > 2: % presumably, a user never tries to intersect no paths
    errhelp "I'll ignore superfluous paths.";
    errmessage "PX: too many paths (" & decimal(i*) & ")";
  fi
  auto* := unknown q2; if auto*: q2 = q1; fi;
  intersect_curves(auto*, q*);
  if qnum = 0: % emergency?
    if not path r[0]: scantokens("path_" & generisize(str r) & "[]"); fi
    if not numeric rnum: scantokens("numeric_" & generisize(str rnum)); fi
    if auto*: rnum := -1; r1 := q1*;
    else: rnum := -2; r1 := q1*; r2 := q2*; fi
  else: recombine(auto*, q*, r); fi
enddef;

```

### A MODULE THAT FINDS AN ENVELOPE OF A PATH BEING DRAWN WITH AN ELLIPTICAL OR A ONE-DIMENSIONAL (“RAZOR”) PEN

The following macros approximate the envelope of an elliptical or a razor pen. The exact solution is impossible—in general, the envelope is not a Bézier curve, therefore some heuristics is, in general, unavoidable. We assumed that the backbone of a figure is such that the envelope does not form loops at smoothly joined nodes. Moreover, all Bézier segments appearing in the process **should not** contain inflection points (the reason for this limitation is the method of finding an approximation of a pen envelope). If the latter condition is not fulfilled, one may expect weird results (see the usage of the ... operator in the code of *pen\_stroke\_edge*).

### MODUŁ ZNAJDUJĄCY OBRYS OBSZARU ZAMALOWYWANEGO PRZEZ PIÓRKO ELIPTYCZNE LUB JEDNOWYMIAROWE („ŻYŁETKOWE”)

Poniższe makra przybliżają brzeg ścieżki kreślonej eliptycznym lub „żyłetkowym” piórkiem. Dokładne rozwiązanie nie jest możliwe – w ogólności brzeg nie jest krzywą Béziera, zatem heurystyki w ogólności uniknąć się nie da. Założyliśmy, że szkielet figury jest tak skonstruowany, że przy gładko połączonych węzłach krawędź piórka nie rysuje pętli. Ponadto krzywe pojawiające się w trakcie przetwarzania **nie powinny mieć** punktów przegięcia (co wiąże się z zastosowanym sposobem aproksymacji obrysu piórka). Jeśli to założenie nie jest spełnione, można oczekiwać dziwacznych efektów (p. użycie operatora ... w kodzie makra *pen\_stroke\_edge*).

We assume that slanting should not distort a pen. Therefore, if a glyph is to be slanted *after* expanding a stroke, which usually is the case, the envelope should be constructed with an *unslanted pen*. Macros **slant\_stroke**, **unslant\_stroke**, and *unslant\_angle* are devised to facilitate handling this situation. These macros refer to the variable *slant\_stroke\_val*; it should be assigned a definite value prior to expanding stroke.

Zakładamy, że pochylenie (*slanting*) nie powinno wpływać na kształt piórka. Zatem jeśli obrys konstruujemy w taki sposób, że pochylenie jest wykonywane po wyznaczeniu obrysu piórka, co zwykle ma miejsce, to obrys powinien być znajdowany dla piórka poddanego pochyleniu odwrotnemu (*unslanting*). Makra **slant\_stroke**, **unslant\_stroke** oraz *unslant\_angle* zostały wprowadzone po to, aby ułatwić obsługę takiego zabiegu. Zmiennej *slant\_stroke\_val*, do której te makra się odwołują, należy nadać stosowną wartość przed wyznaczaniem obrysu piórka.

**def slant\_stroke =**

**if known *slant\_stroke\_val*: slanted****slant\_stroke\_val fi**  
**enddef;**

**def unslant\_stroke =**

**if known *slant\_stroke\_val*: slanted – *slant\_stroke\_val* fi**  
**enddef;**

**vardef *unslant\_angle*(expr *a*) = angle(dir(*a*) **unslant\_stroke**) enddef;**

Macro *fix\_nib* returns a path. If *y\_diam* parameter is 0, a “razor” pen (a segment) is returned, otherwise it is an approximation of an ellipse. We do our best to avoid unnecessary nodes, hence the approximation is somewhat complicated; another reason for the complication is that interpolation and affine transformations do not commute, therefore the appropriate nodes are found for the untransformed pen, and only then the pen is transformed. *Note*: So far, there is no explicit relation between a built-in METAPOST pen mechanism and the *fix\_nib* operation, in particular, **beginfig** does not alter the setting of *default\_nib*. Needs rethinking.

Makro *fix\_nib* zwraca ścieżkę. Jeśli parametr *y\_diam* jest równy 0, zwracane jest piórko „żyłetkowe” (odcinek), w przeciwnym razie wynikiem jest (przybliżona) elipsa. Ponieważ chcemy w miarę możliwości unikać zbędnych węzłów, konstrukcja przybliżenia jest nieco skomplikowana. Dodatkowym czynnikiem komplikującym jest to, że interpolacja nie jest przemiana względem przekształceń afinicznych, dlatego najpierw wyznaczane są stosowne węzły na niezdeformowanym piórku, a dopiero potem piórko jest przekształcane. *Uwaga*: Nie ma bezpośredniego powiązania między operacją *fix\_nib* a wbudowanym w METAPOST-a mechanizmem piórek, w szczególności **beginfig** nie zmienia ustawienia *default\_nib*. Rzecz wymaga przemyślenia.

```

vardef fix_nib(expr x_diam, y_diam, rot_angle) =
  if (x_diam <> 0) and (y_diam <> 0): fix_elliptic_nib(x_diam, y_diam, rot_angle)
  elseif (x_diam <> 0) and (y_diam = 0): fix_razor_nib(x_diam, rot_angle)
  elseif (x_diam = 0) and (y_diam <> 0): fix_razor_nib(y_diam, rot_angle + 90)
  else:
    errhelp "I'll use the default pen, but I'd suggest to cancel the job.";
    errmessage "PX: the null pen is not allowed";
    default_nib
  fi
enddef;

vardef fix_razor_nib(expr x_diam, rot_angle) =
  ((-1/2 x_diam, 0) -- (1/2 x_diam, 0)) rotated rot_angle unslant_stroke
enddef;

vardef fix_elliptic_nib(expr x_diam, y_diam, rot_angle) =
  save p*; path p*;
  % construct a temporary ellipse:
  p* := fullcircle
  xscaled x_diam yscaled y_diam rotated rot_angle unslant_stroke;
  % construct an elliptic pen path having
  % 4 or, if necessary (heuristic), 6 nodes:
  (for d = up unslant_stroke, left,
    if (y_diam/x_diam < 1/2) and (abs(rot_angle mod 90) > 5):
      left rotated rot_angle unslant_stroke,
    fi
    down unslant_stroke, right,
    if (y_diam/x_diam < 1/2) and (abs(rot_angle mod 90) > 5):
      right rotated rot_angle unslant_stroke
    fi:
    (point(directiontime d of p*) of fullcircle)
    {direction(directiontime d of p*) of fullcircle} ...
  endfor cycle) xscaled x_diam yscaled y_diam rotated rot_angle unslant_stroke
enddef;

```

Arcs of a pen shorter than *ignore\_nib\_limit* will be joined together to form larger ones. Remember to adjust the parameter *ignore\_nib\_limit* if the size of *default\_nib* is significantly changed.

Łuki piórka krótsze niż *ignore\_nib\_limit* zostaną połączone w większe segmenty. Należy pamiętać o zmianie parametru *ignore\_nib\_limit* przy znaczącej zmianie rozmiaru piórka (*default\_nib*).

```
newinternal ignore_nib_limit; ignore_nib_limit := 5;
```

```
path default_nib;
```

```
default_nib := fix_nib(50, 50, 0); % hundred times as large as a default plain pen
```

```
newinternal default_elongation, default_join, default_cap;
```

```
default_elongation := 1/2;
```

```
default_join := 1;
```

```
% 0 -- tip, default elongation used
```

```
% 1 -- pen join, default elongation ignored
```

```
% 2 -- tip, default elongation ignored, elongation=0 used
```

```
default_cap := 1;
```

```
% 0 -- cut 90 rel
```

```
% 1 -- pen end
```

*tangent\_point*, *pen\_join*, *pen\_stroke\_edge*\*, and *pen\_stroke\_edge* are auxiliary macros, exploited by the main macro, i.e., **pen\_stroke**.

*tangent\_point*, *pen\_join*, *pen\_stroke\_edge*\* i *pen\_stroke\_edge* to makra pomocnicze, użyte w głównym makrze, tj. **pen\_stroke**.

```
vardef tangent_point(expr d, nib) = % d -- direction of pen movement
```

```
save a*;
```

```
point if cycle nib: (directiontime d of nib) else:
```

```

    hide( $a^* := \text{turn\_ang}(d, (\text{point } 1 \text{ of } nib) - (\text{point } 0 \text{ of } nib))$ )
    if  $\text{abs}(a^* \bmod 180) < .1: \frac{1}{2} \% \text{ emergency}$ 
    elseif  $a^* < 0: 0$  else: 1 fi
  fi of nib
enddef;
%
vardef pen_join(expr a, b, c, nib) =
  % deleting superfluous nodes is based on the arclength operation
  % which, obviously, is not preserved after slanting, but let's hope
  % it does not matter (too much)
  save  $t^*, m^*, m^{**}, ta^*, tb^*, p^*$ ; path  $p^*$ ;
   $m^* := \text{infinity}$ ; % will be the minimal length of nib's segment
  for  $t^* := 0$  upto  $\frac{1}{2} \text{ length}(nib) - 1$ :
     $m^{**} := \text{arclength}(\text{subpath}(t^*, t^* + 1) \text{ of } nib)$ ;
    if  $m^{**} < m^*$ :  $m^* := m^{**}$ ; fi
  endfor
  if  $m^* < \text{ignore\_nib\_limit}$ :
    message "PX: the shortest nib segment < ignore\_nib\_limit (" &
      decimal( $m^{**}$ ) & "<" & decimal( $\text{ignore\_nib\_limit}$ ) & ")";
    fi
     $p^* = nib$  shifted c;
    if cycle nib:
       $ta^* = \text{directiontime } a \text{ of } p^*$ ;  $tb^* = \text{directiontime } b \text{ of } p^*$ ;
       $p^* := \text{pos\_subpath}(ta^*, tb^*) \text{ of } p^*$ ;
      if  $\text{arclength}(p^*) > \text{ignore\_nib\_limit}$ :
        for  $i^* := 0, 0$ :
           $p^* := \text{reverse } p^*$ ; % short segments may appear at both ends
          if  $\text{length}(p^*) > 1$ : % optimization
            if  $\text{arclength}(\text{subpath}(0, 1) \text{ of } p^*) < \frac{1}{4} \text{ ignore\_nib\_limit}$ :
              % cf. the comment concerning  $\frac{1}{4} \text{ ignore\_nib\_limit}$  in
              % pen_stroke_edge below
               $p^* := (\text{point } 0 \text{ of } p^*) \dots \text{controls}(\text{postcontrol } 1 \text{ of } p^*) \text{ and}$ 
                 $(\text{precontrol } 2 \text{ of } p^*) \dots \text{subpath}(2, \text{infinity}) \text{ of } p^*$ ;
            fi
          fi
        endfor
      else:
         $p^* := (\text{point } 0 \text{ of } p^*)\{a\} \dots \{b\}(\text{point } \text{length}(p^*) \text{ of } p^*)$ ;
      fi
    else: % razor nib
       $p^* := \text{tangent\_point}(a, p^*) \text{ -- } \text{tangent\_point}(b, p^*)$ ;
    fi
     $p^*$ 
  enddef;

```

The finding of a pen envelope for a given Bézier segment, defined by nodes  $a$ ,  $b$ ,  $c$ , and  $d$ , begins with the placing the pen at the ends of the Bézier segment (i.e., at the points  $a$ ,  $d$ ) and finding the corresponding points  $a'$  and  $d'$  where the pen outline is parallel to the direction of the original path at these points. Then, the outline is constructed. For *pen\_stroke\_method* = 0 (default), the envelope segment is constructed by setting the beginning and final directions (optionally, the direction at a given node can be ignored); for *pen\_stroke\_method* = 1 or 2 an alternative (more elaborate) procedure is involved which explicitly computes control nodes  $b'$  and  $c'$  of the resulting path basing on a heuristic assumption that  $\text{length}(b' - a') / \text{length}(b - a) \approx \text{length}(c' - d') / \text{length}(c - d) \approx \text{length}(a' - d') / \text{length}(a - d)$ . The default method never produce concave edges because the operator  $\dots$  is used always; the alternative methods employs the operator *force\_convex\_edge* instead; for *pen\_stroke\_method* = 1 the convex edges are forced (i.e, inflexion points are being removed), for *pen\_stroke\_method* = 2 no forcing of convex edges takes place.

Wyznaczanie brzegu obszaru rysowanego piórkem dla segmentu Béziera określonego przez węzły  $a$ ,  $b$ ,  $c$  i  $d$ , rozpoczyna się od umieszczenia piórka w węzłach końcowych (tj.  $a$  i  $d$ ) znalezienia odpowiadających im punktów  $a'$  i  $d'$  w których obwiednia piórka jest równoległa do oryginalnej ścieżki. Następnie konstruowana jest krawędź brzegu obszaru rysowanego piórkem. Dla metody domyślnej (*pen\_stroke\_method* = 0) ścieżka jest budowana na podstawie kierunku początkowego i końcowego. Dla metody alternatywnej (*pen\_stroke\_method* = 1 lub 2) ścieżka jest wyznaczana przez jawne wyliczenie naciągów ( $b'$  i  $c'$ ) na podstawie heurystycznego założenia  $\text{length}(b' - a') / \text{length}(b - a) \approx \text{length}(c' - d') / \text{length}(c - d) \approx \text{length}(a' - d') / \text{length}(a - d)$ . Pierwsza z metod (tzn. domyślna) nie daje w wyniku ścieżki wklęsłej, gdyż używany jest zawsze operator  $\dots$ ; w wypadku metody alternatywnej stosowany jest operator *force\_convex\_edge*, który wymusza wypukłość (tzn. usuwa punkty przegięcia) dla *pen\_stroke\_method* = 1, natomiast nie wymusza dla *pen\_stroke\_method* = 2.

```

vardef extrapoline expr  $t$  of  $B$  =% the result may be not a single segment
  save  $l^*, t^*$ ;
   $(t_a^*, t_b^*) = t$ ; %  $0 \leq ta^* < tb^* \leq 1$ !
   $l^* = \text{arclength}(B) / (t_b^* - t_a^*)$ ;  $l_a^* = l^* * t_a^*$ ;  $l_b^* = l^* * (1 - t_b^*)$ ;
  if  $t_a^* > 0$ : ((point 0 of  $B$ ) -  $l_a^* * (\text{upostdir}$  0 of  $B$ )) -- fi
     $B$ 
  if  $t_b^* < 1$ : -- ((point 1 of  $B$ ) +  $l_b^* * (\text{upredir}$  1 of  $B$ )) fi
enddef;
%
vardef force_convex_edge(expr  $za, zb, zc, zd$ ) =
  save  $a^*, b^*, c^*, d^*, z^*$ ;
   $a^* := \text{length}(zd - za)$ ;  $b^* := \text{length}(zb - za)$ ;  $c^* := \text{length}(zc - zb)$ ;  $d^* := \text{length}(zd - zc)$ ;
  if ( $-a^* + b^* + c^* + d^* > a^* / \text{infinity}$ ):
    if pen_stroke_method = 2:
       $za \dots \text{controls } zb \text{ and } zc \dots zd$ 
    else:
      if ( $a^* > 0.01$ ) and ( $b^* > 0.01$ ) and ( $c^* > 0.01$ ) and ( $d^* > 0.01$ ): % no degeneration...
         $a^* := \text{signum}((za - zd) \text{ rotated } -90 \text{ dotnorm}(zb - za))$ ;
         $b^* := \text{signum}((zb - za) \text{ rotated } -90 \text{ dotnorm}(zc - zb))$ ;
         $c^* := \text{signum}((zc - zb) \text{ rotated } -90 \text{ dotnorm}(zd - zc))$ ;
         $d^* := \text{signum}((zd - zc) \text{ rotated } -90 \text{ dotnorm}(za - zd))$ ;
        if ( $(a^* <> b^*)$  or ( $b^* <> c^*)$ ) and ( $a^* = d^*$ ):
          numeric  $b^*, c^*$ ; pair  $z^*$ ;
           $z^* = b^*[za, zb] = c^*[zd, zc]$ ;
           $za \dots \text{controls}$ 
            if  $b^* < 1$ :  $z^*$  else:  $zb$  fi and if  $c^* < 1$ :  $z^*$  else:  $zc$  fi
           $\dots zd$ 
        else:
           $za \dots \text{controls } zb \text{ and } zc \dots zd$ 
      fi
    fi
  fi

```



```

    else:
        za .. controls zb and zc .. zd
    fi
fi
else:
    za -- zd
fi
enddef;
%
vardef pen_stroke_edge*(expr b, b_nib, e_nib)=% b -- Bézier segment
    save pa*, pb*, qa*, qb*, ra*, rb*, sa*, sb*;
    pair pa*, pb*, qa*, qb*, ra*, rb*, sa*, sb*;
    pa* = point 0 of b; ra* = (postcontrol 0 of b) - pa*; sa* = postdir 0 of b;
    pb* = point 1 of b; rb* = (precontrol 1 of b) - pb*; sb* = predir 1 of b;
    qa* = pa* + tangent_point(sa*, b_nib);
    qb* = pb* + tangent_point(sb*, e_nib);
    if pen_stroke_method = 0:
        qa*{sa*} ... {sb*}qb*
    elseif (pen_stroke_method = 1) or (pen_stroke_method = 2):
        save lp*, lq*; lp* = length(pb* - pa*); lq* = length(qb* - qa*);
        if 2lp* < lq*: % heuresis -- too close nodes
            qa*{sa*} ... {sb*}qb*
        else:
            force_convex_edge(qa*, qa* + lq*/lp* * ra*, qb* + lq*/lp* * rb*, qb*)
        fi
    else:
        errhelp "Only the values 0, 1 and 2 for 'pen_stroke_method' are admissible." &
            "Better stop now.";
        errmessage "PX: unknown pen_stroke_method (" &
            decimal(pen_stroke_method) & ")";
    fi
enddef;
%
vardef pen_stroke_edge_@#(expr p) =
    save e*, l*, i*, i**; path e*[ ];
    l* := length(p);
    for i* := 0 upto l* - 1:
        e*[i*] = pen_stroke_edge*(subpath(i*, i* + 1) of p,
            % local_nib_@#(i*), local_nib_@#(i* + 1)); % a nasty bug removed 20.08.2009
            local_nib_@#(i*), local_nib_@#(i* + 1) if cycle p: mod l* fi);
    endfor
    for i* := 0 upto l* if cycle p: -1 else: -2 fi:
        i** := (i* + 1) mod l*;
        save t*;
        t* := turn_ang(predir 1 of e*[i*], postdir 0 of e*[i**]);
        if if known t*: abs(t*) > 1 else: false fi:
            save t*; (t_a, t_b) = e*[i*] intersectiontimes e*[i**];
            if t_a > 0:
                e*[i*] := subpath(0, t_a) of e*[i*];
                e*[i**] := subpath(t_b, 1) of e*[i**];
            elseif known local_tip_@#(i**):
                save tx*, ty*, b*, b**, ei*, ei**, ei*[ ], ei**[ ];
                (tx*, ty*) = local_tip_@#(i**);
                ei* := if is_line(e*[i*]):
                    (point 0 of e*[i*]) --
                    (1/ abs(tx*)) [point 0 of e*[i*], point 1 of e*[i*]]
                elseif tx* < 0: hide(b* := 1) extrapoline(0, abs(tx*)) of e*[i*]
                else: extrapolate(0, abs(tx*)) of e*[i*] fi;

```

```

    ei** := if is_line(e*[i**]):
      (1/(1 - abs(ty**)))[point 1 of e*[i**], point 0 of e*[i**]] --
      point 1 of e*[i**]
    elseif ty* < 0: hide(b** := 1)extrapoline(abs(ty*), 1) of e*[i**]
    else: extrapolate(abs(ty*), 1) of e*[i**] fi;
% clumsy HEURESIS (choosing an optimal intersection point, if there are
% more intersections):
save t*; (t*a1, length(ei**) - t*b1) = ei* intersectiontimes reverse ei**;
if t*a1 > 0:
  ei*1 := if (known b*) and (t*a1 > 1):
    force_convex_edge(point 0 of e*[i*], postcontrol 0 of e*[i*],
      precontrol 1 of e*[i*], point t*a1 of ei*)
  else: subpath(0, t*a1) of ei* fi;
ei**1 := if (known b**) and (t*b1 < 1):
  force_convex_edge(point t*b1 of ei**, postcontrol 0 of e*[i**],
    precontrol 1 of e*[i**], point 1 of e*[i**])
  else: subpath(t*b1, infinity) of ei** fi;
(length(ei*) - t*a2, t*b2) = reverse ei* intersectiontimes ei**;
if length((t*a1, t*b1) - (t*a2, t*b2)) > eps:
  ei*2 := if (known b*) and (t*a2 > 1):
    force_convex_edge(point 0 of e*[i*], postcontrol 0 of e*[i*],
      precontrol 1 of e*[i*], point t*a2 of ei*)
  else: subpath(0, t*a2) of ei* fi;
ei**2 := if (known b**) and (t*b2 < 1):
  force_convex_edge(point t*b2 of ei**, postcontrol 0 of e*[i**],
    precontrol 1 of e*[i**], point 1 of e*[i**])
  else: subpath(t*b2, infinity) of ei** fi;
if arclength(ei*1) + arclength(ei**1) > arclength(ei*2) + arclength(ei**2):
  ei*1 := ei*2; ei**1 := ei**2;
fi
fi
e*[i*] := ei*1; e*[i**] := ei**1;
fi
fi
fi
endfor
for i* := 0 upto l* - 1:
  hide(i** := (i* - 1) mod l*)
  if cycle p or (i* > 0):
    if length((point 1 of e*[i**]) - (point 0 of e*[i*])) > 1/4 ignore_nib_limit:
      % the constant 1/4 ignore_nib_limit plays a similar role
      % to that of the SNAP-TO-NODE variable in pf2mt1.awk
      (point 1 of e*[i**])
    if known local_tip@#(i*): -- else:
      && pen_join(predir 1 of e*[i**], postdir 0 of e*[i*], point i* of p,
        local_nib@#(i*))&&
    fi
  fi
fi
% reconstruct e*[i*] (possibly ignoring direction(s)):
(point 0 of e*[i*])
if is_line(e*[i*]):
  % the using of -- circumvents METAFONT/METAPOST instable behaviour:
  % the operator ... may cause that a control point and a node
  % (nearly) coincide (note that this is feature, not a bug);
  % thus, it is advisable for pen_stroke_method = 0; supposedly,
  % it is also adequate for pen_stroke_method = 1:
  --

```

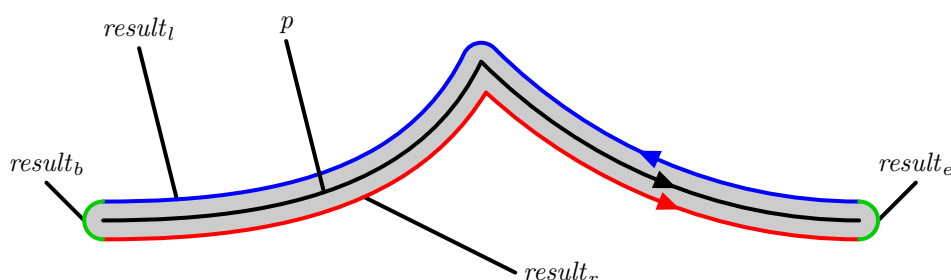
```

else:
  if pen_stroke_method = 0:
    if not ignore_dir*(i*): {postdir 0 of e*[i*]} fi ...
    if not ignore_dir*(i* + 1): {predir 1 of e*[i*]} fi
  elseif (pen_stroke_method = 1) or (pen_stroke_method = 2):
    .. controls(postcontrol 0 of e*[i*]) and (precontrol 1 of e*[i*]) ..
  fi
fi
endifor
if cycle p: cycle else: (point 1 of e*[l* - 1]) fi
enddef;
newinternal pen_stroke_method;

```

Macro **pen\_stroke** performs an operation known as “expanding stroke”; we’ll call the result of the operation a “pen envelope” (for a given path). The macro has one optional parameter, *opts* (**text**), and two obligatory ones: input path *p* (**expr**) and a *result* (**suffix**). A user has an access to subpaths of the envelope, namely: *result<sub>r</sub>* is the right edge of the envelope, *result<sub>l</sub>*—its left edge, *result<sub>b</sub>*—is a fragment of the pen outline joining left and right edge of the envelope at the beginning node of the path, *result<sub>e</sub>*—is a similar fragment at the ending node of the path (see the picture below). If the path *p* is cyclic, then *result<sub>e</sub>* and *result<sub>b</sub>* are undefined, otherwise the variable *result* contains additionally the complete expanded stroke.

Makro **pen\_stroke** realizuje operację znaną jako „ekspansja ścieżki” (*expanding stroke*); wynik tej operacji będziemy określać mianem „obrysu piórka” (dla danej ścieżki). Makro **pen\_stroke** ma jeden parametr opcjonalny *opts* (typu **text**) oraz dwa parametry obowiązkowe: ścieżkę wejściową *p* (typu **expr**) i wynik *result* (typu **suffix**). Użytkownik ma dostęp do podścieżek obrysu, mianowicie: *result<sub>r</sub>* to prawa krawędź obrysu, *result<sub>l</sub>* – lewa, *result<sub>b</sub>* – fragment obwiedni piórka łączący lewą i prawą krawędź w początkowym punkcie ścieżki, *result<sub>e</sub>* – analogiczny fragment na końcu ścieżki (p. rysunek poniżej). Jeśli ścieżka *p* jest zamknięta, to *result<sub>e</sub>* oraz *result<sub>b</sub>* są niezdefiniowane, jeśli otwarta – zmienna *result* zawiera dodatkowo kompletny obrys ścieżki.



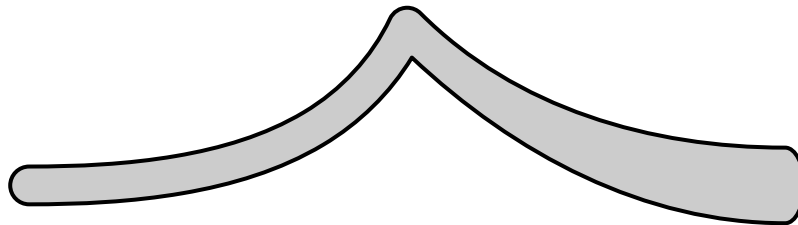
For finding an envelope, a default path (*default\_nib*, returned by *fix\_nib*) is used except nodes for which the parameter *opts* sets another pen. Mastering the usage of the parameter *opts* allows a user to achieve nontrivial effects. The parameter *opts* is a list (space-separated or semicolon-separated) of the following operators: (1) *nib*, (2) *cut*, (3) *tip*, and (4) *ignore\_directions*.

Do wyznaczenia obrysu wykorzystywane jest piórko *default\_nib* (zdefiniowane za pomocą makra *fix\_nib*), za wyjątkiem węzłów, dla których parametr *opts* definiuje inne piórko. Biegłe wykorzystanie parametru *opts* pozwala na uzyskiwanie nietrywialnych efektów. Parametr *opts* jest listą następujących poleceń (separowaną spacjami lub średnikami, jak kto woli): (1) *nib*, (2) *cut*, (3) *tip* oraz (4) *ignore\_directions*.

Ad 1. The macro *nib* has two parameters: *nib*(pen)(list\_of\_nodes), where “pen” is a path returned by macro *fix\_nib*, and “list\_of\_nodes” contains comma-separated numbers (times) of the nodes of the path *p* at which a given pen is to be used. If needed, the outline is complemented at corner nodes with a fragment of a pen path. Such a join corresponds to the setting *linejoin* := *rounded* in METAPOST. If the path *p* is non-cyclic, its ends are also complemented with appropriate fragments of a pen path (the setting *linecap* := *rounded*). Such a method of joining is also applied by **pen\_stroke** to nodes not mentioned in the parameter *opts*. The result of the following statement

**pen\_stroke**(*nib*(*default\_nib* xyscaled (1, 2))(infinity))(p)(q)

that changes the pen at the last node of the path, is shown in the following picture:



Ad 2. The call of the macro *cut* has the form: *cut*(angle, pen)(list\_of\_nodes) or *cut*(pen, angle)(list\_of\_nodes), where “pen” and “list\_of\_nodes” are defined as previously. The pen parameter can be omitted which means using a default pen (*default\_nib*). The macro replaces a default pen with a special “razor” pen at specified nodes. More precisely, it is a projection of a given pen in the direction of the path *p* at a given node onto a straight line going through this node under the angle specified in the respective parameter of the macro. Ufff. . . The angle of the straight line can be defined either absolutely (with respect to the axis *x*) or—by adding a prefix ‘rel’—relatively to the direction of the path at a given node. From the point of view of a user, the result of the macro *cut* is “cutting” the expanded stroke with a straight line. This operation is particularly useful at the ends of a path and corresponds to the setting *linecap* := *butt* in METAPOST, except that in METAPOST one cannot specify angles. The result of the statement

**pen\_stroke**(*cut*(45)(0) *cut*(*default\_nib* xyscaled (1, 2), rel 90)(infinity))(p)(q)

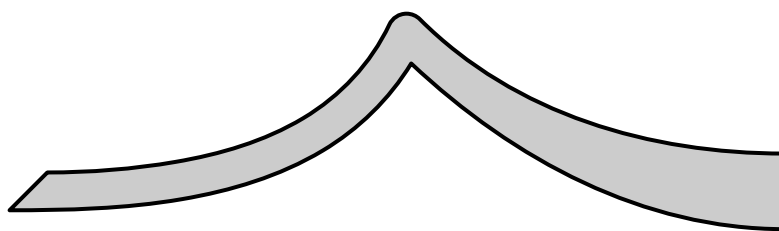
that cuts both ends and, moreover, changes a pen at the ending node is shown in the figure below (at the beginning node, the absolute angle of 45 degrees is specified, at the ending one—the relative angle of 90 degrees):

Ad 1. Wywołanie makra *nib* ma postać *nib*(piórko)(lista\_węzłów), gdzie „piórko” jest ścieżką zdefiniowaną za pomocą makra *fix\_nib*, a „lista\_węzłów” zawiera rozdzielone przecinkami numery (czasy) węzłów ścieżki *p*, w których dane piórko ma być użyte. W punktach narożnych obrys jest w razie potrzeby uzupełniany fragmentem ścieżki piórka. Odpowiada to METAPOST-owemu ustawieniu *linejoin* := *rounded*. W przypadku ścieżki otwartej także jej końce są uzupełnianie fragmentem piórka (*linecap* := *rounded*). Taki sposób łączenia stosowany jest również przez makro **pen\_stroke** w węzłach niewymienionych w parametrze *opts*. Wynik użycia polecenia

powodującego zmianę piórka w ostatnim węźle ścieżki przedstawia poniższy rysunek:

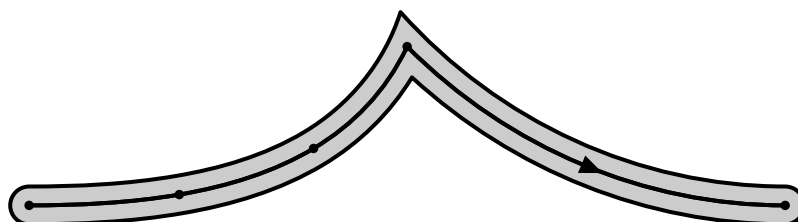
Ad 2. Wywołanie makra *cut* ma postać *cut*(kąt, piórko)(lista\_węzłów) lub *cut*(piórko, kąt)(lista\_węzłów), gdzie „piórko” i „lista\_węzłów” są zdefiniowane jak wyżej. Parametr określający piórko można pominąć, co oznacza użycie piórka domyślnego (*default\_nib*). Makro powoduje użycie w zadanych punktach specjalnego piórka „żyłkowego”, będącego rzutem danego piórka w kierunku zgodnym z kierunkiem ścieżki *p* w danym węźle na prostą, przechodzącą przez ten węzeł pod kątem określonym przez parametr „kąt”. Ufff. . . Kąt prostej może być określany względem osi *x*, lub – po dodaniu przedrostka ‘rel’ – względem kierunku ścieżki w danym węźle. Z punktu widzenia użytkownika efektem działania makra *cut* jest „przecięcie” obwiedni w danym węźleadaną prostą – jest to użyteczne głównie na końcach ścieżki. Makro to odpowiada METAPOST-owemu ustawieniu *linecap* := *butt*, z tym że METAPOST nie pozwala na dowolne zadanie kąta ścięcia. Wynik użycia polecenia

definiującego ścięcia na końcach ścieżki wraz ze zmianą piórka w punkcie końcowym przedstawia poniższy rysunek (w punkcie początkowym zadany jest kąt bezwzględny 45 stopni, w końcowym – 90 stopni względem kierunku ścieżki):



Ad 3. The call of the macro *tip* has the form *tip*(pen, pre\_elongate, post\_elongate)(list\_of\_nodes), where “pen” and “list\_of\_nodes” have the same meaning as previously. In particular, a pen can be omitted. At corner nodes specified in the list of nodes, the consecutive elements of the outline are not joined with an appropriate subpath of a pen; instead, they are elongated (extrapolated) until they intersect. This process corresponds (roughly) to the METAPOST setting *linejoin := mitered*:

Ad 3. Wywołanie makra *tip* ma postać *tip*(piórko, przed\_wydłużenie, po\_wydłużenie)(lista\_węzłów), gdzie „piórko” i „lista\_węzłów” mają postać jak w poprzednich makrach, w szczególności piórko również można pominąć. W węzłach narożnych, wyspecyfikowanych za pomocą tego makra, sąsiednie segmenty nie są łączone fragmentem ścieżki piórka, tylko są przedłużane (ekstrapolowane) i jest znajdowany ich punkt przecięcia. Procedura ta odpowiada z grubsza METAPOST-owemu ustawieniu *linejoin := mitered*:



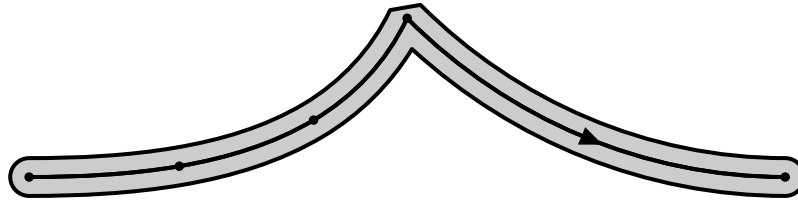
The illustration above is the result of the following call of the macro **pen\_stroke** (the macro *tip* is invoked with default settings, only the number of a node is specified):

Powyższa ilustracja to rezultat wywołania makra **pen\_stroke** w następujący (użycie makra *tip* bazuje na ustawieniach domyślnych, wyspecyfikowany jest jedynie numer węzła) sposób:

**pen\_stroke**(*tip*)(3))(p)(q); **draw** q;

The optional parameters *pre\_elongation* and *post\_elongation* define how far the consecutive edges (segments) should be elongated in order to make them intersect each other (the measure is the time). If one parameter is omitted, both will receive the same value; if both are omitted, a default value, (0.5, 0.5) (it corresponds to elongation by circa 50%), will be used. The precise meaning of the pre- and post-elongation is defined as follows: for a given pre-edge  $e_1$ , post-edge  $e_2$ , pre-elongation  $v_1$  and post-elongation  $v_2$ , the paths **extrapolate**(0,  $1/(1+v_1)$ ) of  $e_1$  and **extrapolate**( $v_2/(1+v_2)$ , 1) of  $e_2$  are computed (i.e., for the default elongation: **extrapolate**(0,  $2/3$ ) of  $s_1$  and **extrapolate**( $1/3$ , 1) of  $s_2$ , respectively). If elongated curves do not intersect, the terminal nodes of the consecutive segments are joined with a straight line. The latter property can be used to obtain a result corresponding to the METAPOST setting *linejoin := beveled*: it suffices to apply a null elongation, i.e., *tip*(0)(list\_of\_nodes). Changing the first (empty) parameter of the *tip* macro in the previous example would yield the following result:

Parametry opcjonalne „przed\_wydłużenie” i „po\_wydłużenie” określają, jak daleko mają być przedłużane krawędzie (segmenty) – miarą jest tu czas – aby można było wyznaczyć ich punkt przecięcia. W przypadku pominięcia jednego z tych parametrów oba otrzymują tę samą wartość; jeśli oba zostaną pominięte, użyta będzie wartość domyślna, tj. (0.5, 0.5), odpowiadająca wydłużeniu mniej więcej o 50%. Ściśle biorąc: dla danej „przed-krawędzi”  $e_1$ , „po-krawędzi”  $e_2$ , „przed-wydłużenia”  $v_1$  oraz „po-wydłużenia”  $v_2$  wyznaczane są ścieżki **extrapolate**(0,  $1/(1+v_1)$ ) of  $e_1$  oraz **extrapolate**( $v_2/(1+v_2)$ , 1) of  $e_2$  (tzn. dla wydłużenia domyślnego odpowiednio **extrapolate**(0,  $2/3$ ) of  $s_1$  oraz **extrapolate**( $1/3$ , 1) of  $s_2$ ). Jeśli przedłużenia nie przetną się, węzły obrysu łączone są odcinkiem prostym. Ostatnia własność umożliwia osiągnięcie efektu analogicznego do skutków ustawienia *linejoin := beveled*. W tym celu należy użyć zerowego wydłużenia: *tip*(0)(lista\_węzłów). Użycie w poprzednim przykładzie zera zamiast pierwszego (pustego) parametru makra *tip* dałoby następujący efekt:



Ad 4. The macro *ignore\_directions* has a different character. It is invoked with one parameter being a comma-separated list of nodes:

*ignore\_directions*(list\_of\_nodes). The numbers *must* be followed by suffixes *l* or *r*. The macro causes that, at specified nodes, the direction of the outline is not forced to be parallel to the direction of the path *p* (which is the default); instead, the direction is calculated by METAPOST. Suffixes determine whether the direction is not to be forced at the right (*r*) or the left (*l*) edge (with respect to the direction of the path *p*). This heuristic trick can be used to improve the appearance of the outline if the “inner” part of the envelope has too tight arcs.

Ad 4. Wywołanie makra *ignore\_directions* ma postać *ignore\_directions*(lista\_węzłów), gdzie „lista węzłów” zawiera rozdzielone przecinkami numery (czasy) węzłów ścieżki *p* z obowiązkowymi sufiksami *l* albo *r*. Makro to spełnia nieco inną rolę niż poprzednie – powoduje mianowicie, że w zadanych węzłach kierunek krzywej tworzącej obrys nie jest wymuszany (normalnie jest on równoległy do kierunku ścieżki *p* w odpowiadającym punkcie). Sufiksy określają, czy niewymuszanie kierunku ma dotyczyć prawej (*r*) czy też lewej (*l*) strony obrysu (patrząc zgodnie z kierunkiem ścieżki). Taki heurystyczny zabieg poprawia na ogół wygląd obrysu, gdy jego „wewnętrzna” część ma zbyt ciasne łuki.

```

vardef pen_stroke(text opts)(expr p)(suffix result) =
  forsuffices $ =, r, l, b, e:
    if not path result$: scantokens("path_" & generisize(str result$)); fi
  endfor
  save a*, a**, d*, i*, k*, n*, p*, z*, norm*, norml*, normr*, normlr*,
    fix_opts*, ignore_dir*, ignore_dir**, local_nib*, local_nib**,
    local_tip*, default_tip*, local_tip**, % internal
    all, rel, last, nib, cut, tip, ignore_directions, current_node; % exported
  numeric ignore_dir**[ ]; pair default_tip*, local_tip**[ ];
  path local_nib**[ ];
  pair a*, d*, z*[ ]; path p*;
  vardef norm* primary n =
    if cycle p: n mod last else: if n < 0: 0 elseif n > last: last else: n fi fi
  enddef;
  vardef norml* primary n = - norm* n - 1 enddef;
  vardef normr* primary n = norm* n + 1 enddef;
  vardef normlr* @# primary i = if str @# = "1": - norm*(last - i) - 1 else: i + 1 fi enddef;
  last = length(p);
  vardef rel primary a =
    angle((gendir current_node of p) slant_stroke) + a
  enddef;
  def all =
    hide(% locally we use the prefix rather than postfix notation;
      % a trick due to the suffix parameter of the allcont* macro
      vardef l primary n = (norml* n, 0) enddef;
      vardef r primary n = (normr* n, 0) enddef) allcont*
  enddef;
  def allcont* suffix $ =
    $0 for i* := 1 upto last if cycle p: -1 fi: , $i_ endfor
  enddef;
  vardef fixopts*(suffix optname)(text nodes) text val =
    save l, r, lcont*, rcont*;
    def l = lcont* whatever enddef; primarydef a lcont* b = (norml* a, 0) enddef;
    def r = rcont* whatever enddef; primarydef a rcont* b = (normr* a, 0) enddef;
    for n* := nodes:

```

```

    if numeric n*:
        current_node := norm* n*;
        optname[norml* n*] := optname[normr* n*]
    else:
        current_node := abs(xpart n*) - 1; % the inverse of both norml* and normr*
        optname[xpart(n*)]
    fi := val; % val may depend on current_node
endfor
enddef;
def nib(text nib*)(text nodes)=% nib and node list
    fixopts*(local_nib**)(nodes)
    begingroup
        p* := default_nib; for k* := nib*: p* := k*; endfor p*
    endgroup;
enddef;
def cut(text nib_and_ang)(text nodes)=% angle, nib and node list
    fixopts*(local_nib**)(nodes)
    begingroup
        p* := default_nib;
        for k* := nib_and_ang:
            if numeric k*: a* := dir(unslant_angle(k*)); else: p* := k*; fi
        endfor
        d* := gendir current_node of p;
        z1* := whatever * a* = tangent_point(d*, p*) + whatever * d*;
        z2* := whatever * a* = tangent_point(-d*, p*) + whatever * d*;
        z1* -- z2*
    endgroup;
enddef;
def tip(text nib_and_lim)(text nodes)=% limit(s) and node list
    i* := 0; for n* := nib_and_lim: if numeric n*: i*[incr i*] := n*; fi endfor
    fixopts*(local_tip**)(nodes)
    elongation_to_times(if i* = 0: default_elongation, default_elongation
        elseif i* = 1: i1*, i1* else: i1*, i2* fi);
    fixopts*(local_nib**)(nodes)
    begingroup
        p* := default_nib; for k* := nib_and_lim: if path k*: p* := k*; fi endfor p*
    endgroup;
enddef;
def ignore_directions(text nodes)=% node list
    fixopts*(ignore_dir**)(nodes)1;
enddef;
if default_cap = 0:
    if not cycle p: cut(rel 90)(0, last); fi
elseif default_cap = 1: % do nothing
else:
    errhelp "Admissible_values_are_0,1;_continue,_I'll_use_the_value_1.";
    errmessage "PX:_improper_'default_cap'_value_(" & decimal(default_cap) & ")";
fi
opts;
%
if default_join = 0:
    default_tip* := elongation_to_times(default_elongation, default_elongation);
elseif default_join = 1: % no tip setting, do nothing
elseif default_join = 2:
    default_tip* := (1, 0); % (1, 0) = elongation_to_times(0, 0)
else:
    errhelp "Admissible_values_are_0,1,2;_continue,_I'll_use_the_value_1.";
    errmessage "PX:_improper_'default_join'_value_(" & decimal(default_join) & ")";

```

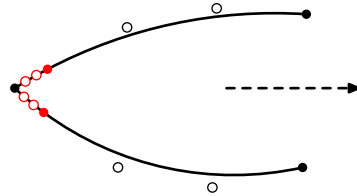
```

fi
vardef ignore_dir*@#(expr i) = known ignore_dir**[normlr* @#i] enddef;
vardef local_tip*@#(expr i) = if known local_tip**[normlr* @#i]:
  local_tip**[normlr* @#i] else: default_tip* fi enddef;
vardef local_nib*@#(expr i) = if known local_nib**[normlr* @#i]:
  local_nib**[normlr* @#i] else: default_nib fi enddef;
resultr := pen_stroke_edger(p);
resultl := pen_stroke_edgel(reverse p);
if not cycle p:
  resultb := pen_cap(predir infinity of resultl, postdir 0 of resultr,
    - postdir 0 of p, point 0 of p, local_nib*l(last), local_nib*r(0));
  resulte := pen_cap(predir infinity of resultr, postdir 0 of resultl,
    predir last of p, point last of p, local_nib*r(last), local_nib*l(0));
  result := resultr && resulte && resultl && resultb && cycle;
fi
enddef;

vardef pen_cap(expr a, b, c, p, niba, nibb) =
  if path_eq(niba, nibb): pen_join(a, b, p, niba)
  else: pen_join(a, c rotated 90, p, niba) - - pen_join(c rotated 90, b, p, nibb)
  fi
enddef;

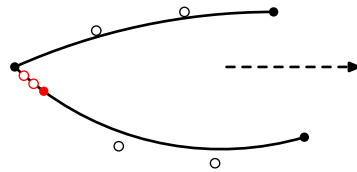
```

Sometimes the results yielded by **find\_outlines** can be further improved, although it is not advisable to rely on fully automatic approach. Macro *correct\_close\_triplets* replaces three close nodes by a single (central) one:



W niektórych przypadkach wyniki generowane przez makro **find\_outlines** można nieco poprawić, aczkolwiek nie należałoby polegać na w pełni automatycznym przetwarzaniu. Makro *correct\_close\_triplets* zastępuje trzy bliskie węzły pojedynczym (centralnym):

while macro *correct\_close\_doublets* replaces two close nodes by a single one (of two nodes of the short segment remains the node at which the change of direction is less abrupt):



natomiast *correct\_close\_doublets* zastępuje dwa bliskie węzły jednym (pozostawiany jest ten koniec krótkiego segmentu, w którym ścieżka zmienia kierunek mniej gwałtownie):

Red colour denotes the points that are being removed; filled circles denote nodes, outlined ones—control points.

Kolor czerwony oznacza punkty usuwane; wypełnione kółka oznaczają węzły, natomiast obrysowane kółka – naciągi.

```

vardef correct_close_triplets(expr p, acc) =
  save i*; i* := 1;
  forever:
    if arclength(subpath(i* - 1, i* + 1) of p) <= acc:
      (point i* of p) ..
      controls(postcontrol incr(i*) of p) and (precontrol incr(i*) of p) ..
    else:
      (point i* - 1 of p) ..
      controls(postcontrol i* - 1 of p) and (precontrol i* of p) ..
    fi
  fi
enddef;

```



```

    fi
    exitif incr( $i^*$ ) = length( $p$ ) + 1;
endfor
(point length( $p$ ) of  $p$ ) & cycle
enddef;
%
vardef correct_close_doublets(expr  $p$ ,  $acc$ ) =
  save  $p^*$ , was_item*; path  $p^*$ ;  $p^* := p$ ;
   $p^* := delete\_nodes(p^*)($ 
    for  $i^* := 0$  upto length( $p$ ) - 1:
      if arclength(subpath( $i^*$ ,  $i^* + 1$ ) of  $p$ ) <=  $acc$ :
        if known was_item*: , fi( $i^*$ , 1)hide(was_item* := 1)
      fi
    endfor);
   $p^*$ 
enddef;
%
endinput

```