EuroTEX '99, Heidelberg

Antykwa Półtawskiego: a parameterized outline font

Bogusław Jackowski, Janusz M. Nowacki,
Piotr Strzelczyk

No doubt, METAFONT is a powerful programming language, well-suited for designing fonts, in many respects much better than popular WYSIWYG programs (precision, possibility of complex constructions, etc.); and, no doubt, there are thousands of fonts used all over the world, only a negligible fraction of them being designed using METAFONT.

1. Computer Modern inheritance

The final exhortation of The METAFONTbook ([1]): "GO FORTH now and create masterpieces of digital typography!" suggests that Donald E. Knuth, when he designed METAFONT, presumed that his idea of parameterized fonts would find many followers. Unfortunately, his expectations failed. Why? The reasons are manifold.

One of the most important aspects is perhaps the exceptional programming talent of Knuth—his style is not so easy to follow. The family of Computer Modern fonts is very intricate: more than hundred files containing nearly a megabyte of METAFONT code. They are rather complex—Knuth modestly admits in The METAFONTbook.

The question arises whether the complexity of the Computer Modern family reflects the nature of the problem (i.e., type design) or rather Knuth's personal traits. We would incline to the latter opinion. A herd of 62 parameters may raise doubts, the more so as they control not only dimensions and slanting, but even the presence of serifs. Questionable also is Knuth's design decision to keep the continuous change of the proportions of glyphs along with the change of font size—it perceivably deteriorates the quality of glyphs in smaller fonts (5–7pt).

Knuth was apparently aware of weak points of the Computer Modern design. In The METAFONTbook he admits: they [the Computer Modern typefaces] were developed rather hastily by the author of the system, who is a rank amateur at such things.

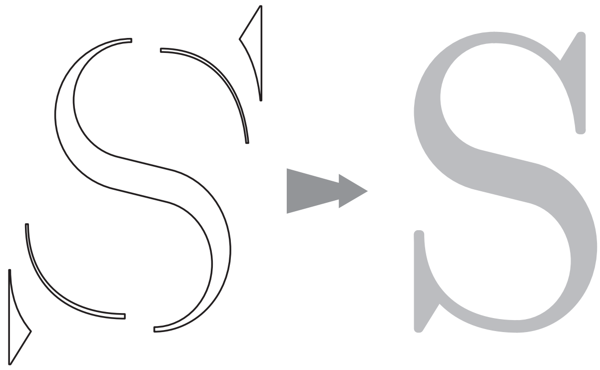

Fig. 1. The construction of the letter 'S' of *cmr10*: five separate elements are stroked with a pen and filled.

Parameterization was certainly a great idea, but it seems that Knuth went too far in exploiting it.

2. Bitmaps versus outlines

Most harmful to the potential success of METAFONT as a tool for type designers was perhaps Knuth's adherence to the bitmap representation of fonts. Although in METAFONT all graphic objects are represented by outlines (Bézier curves), METAFONT's primary task is to generate bitmaps. Therefore the outline representation of glyphs was unimportant for Knuth.

For example, the letter 'S' of *cmr10* consists of five pieces filled and, worse still, stroked with a circular pen (Fig. 1). Many tricks of this kind can be found among the Computer Modern programs: stroking with various pens, erasing (dish serifs), copying bitmaps (German double 'S' in *cmcsc10*), etc.

Such an approach is satisfactory as long as the resulting bitmap is the main concern. The fact is that TEX bitmap fonts have not become a worldwide standard. Instead, outline font formats, *Type 1* (PostScript) and *TrueType* (Windows; its PostScript equivalent is known as *Type 42*), have won the competition.

The problem is that there does not exist a simple method of conversion of METAFONT bitmap-oriented programs into a purely outlined form. Some preliminary results (by Andreĭ Slepukhin, [2]) show that it could be devised specifically for the Computer Modern family, but:

1. the reasonableness of such an effort is doubtful;
2. the converter can be neither efficient nor reliable.

Richard J. Kinch, a staunch devotee of outlines ([4], p. 134) developed an interesting tool for an interactive conversion of Computer Modern fonts to an outline form, *MetaFog* ([3]). Recently, several fonts were prepared by Taco Hoekwater ([5]) using this technology.

In spite of the successes of Kinch's approach, it looks as if re-writing the Computer Modern programs from scratch were more advisable. Still better would be to have a macro package facilitating the *creation* of outline fonts. But is METAFONT the most adequate tool for such a purpose?

3. METAFONT versus METAPOST

In 1989, five years after the first release of METAFONT[1], METAPOST ([6]) came to this world. The originator was John D. Hobby, who designed many of the elegant algorithms employed in METAFONT. Hobby realised that METAFONT is an excellent tool for designing graphics, not only fonts, and that bitmap output is a severe limitation. His idea was to use the METAFONT language to create PostScript output. He did not consider, however, making a tool for generating PostScript fonts. Fortunately, his adaptation was sufficiently general to admit font applications as well.

Again, a question arises: does it make sense to force METAPOST to do things for which it was never intended? The answer is equivocal.

There are some interesting features present in METAFONT and absent from METAPOST, and vice versa. For example, the measuring of arc length is absent from METAFONT and present in METAPOST, whereas METAFONT, but not METAPOST, is capable of measuring the area surrounded by a cyclic path.

From the point of view of the generation of outline fonts, both programs need postprocessing: with METAFONT one has to analyse either a generic font file or a log file; with METAPOST the resulting EPS files are to be processed.

It is intuition that remains in such ambiguous situations—it told us: METAPOST.

[1] Actually, the first version of METAFONT appeared in 1979. Having gathered experience, Knuth released a new version of METAFONT in 1984, re-written form scratch and incompatible with the predecessor. In the source of METAFONT, *mf.web*, the history of METAFONT starts with the statement: Version 0 was completed on July 28, 1984.

ABCDEFGHIJKLŁMN
OPQRSTUVWXYZ
Antykwa Półtawskiego
abcdefghijklłmnopqrstuvwxyz
1234567890

Fig. 2. Antykwa Półtawskiego: letters and digits (lead print).

Fig. 3. Antykwa Półtawskiego: characteristic details and features. Observe polygonal dots, unconventional letters 'g', 'y' and 'w,' and queer right-angled hooks in 'j' and 'f'. Note also the neatly stroked ribbon-shaped diacritical elements of the Polish 'l' and 'L'—certainly, they cannot be called "slashes."

4. Antykwa Półtawskiego

The best method of the verification of intuition is to deal with a real-life case. None of the authors is a professional type designer, so we could not even dream about designing a font *ab ovo*. Fortunately, there exists an elegant typeface, designed in the 'twenties by a Polish typographer, Adam Półtawski (Fig. 2). It was a tempting challenge to test our intuition against Półtawski's typeface.

In the sequel we shall use the Polish name "Antykwa Półtawskiego" instead of the English translation "Półtawski's Antique," as the word "antique"—according to the Collins dictionary—means: a family of typefaces having all lines of nearly equal thickness. This is not true with Półtawski's design. As you can see, it resembles rather neoclassical (also called modern) typefaces.

At first glance, Antykwa Półtawskiego looks very regularly, almost austere. A close inspection, however, reveals many surprising details (Fig. 3).

Studying the imprints of Antykwa Półtawskiego, we found ourselves in the position of the explorers of pyramids: we had to conjecture the rules governing shapes and proportions from faint evidence (or even none). Our situation would improve if we had access to the original Półtawski templates. So far, we have not managed to track them down, but there remains a spark of hope that they exist somewhere.

The bright side of this otherwise confusing situation is that it compels us to looking for a general and flexible approach. Otherwise, we could easily get lost among the peculiarities of Antykwa Półtawskiego.

What we actually expect to obtain is a family of typefaces (not as broad as Computer Modern) resembling Antykwa Półtawskiego. If we have luck, one member of the family may happen to replicate nicely the original Antykwa Półtawskiego design, but it is not crucial. Much more important is the possibility of generating font variations on the fly.

5. A glance at outline paradigms

A compendious survey of technical aspects concerning outline fonts can be found in the Adobe manual concerning *Type 1* fonts ([7]); we summarize here only the most important and general (PostScript-independent) ones.

5.1. Direction of paths

In *Type 1* fonts, paths to be filled and to be unfilled have different orientation: the former—anticlockwise, the latter—clockwise (Fig. 4A). We decided to follow this convention, although neither METAFONT nor METAPOST requires it.

5.2. Overlapping paths

Overlapping paths should be banished forever from outline fonts. It should be noted, however, that many font vendors, including the celebrated Microsoft, distribute fonts that do not follow this fundamental rule (Fig. 4B), which may cause unwanted effects. Therefore the operation of joining two outlines should belong to the set of basic tools for generating outline fonts (cf. Sec. Removing overlaps below).

5.3. Conciseness

The outline of a glyph should contain as few segments as possible (Fig. 4C). For example, consecutive co-linear segments should be replaced by a single one. Users of METAFONT and METAPOST should be aware of the problem of “tiny segments”: joining two paths obtained by an intersection operation (`intersectionpoint` or `interseciontimes`) may lead to nearly coincident and thus superfluous nodes. It is an admittedly subtle problem, yet important from the point of view of the construction of tidy outlines (cf. Sec. Joining paths below).

5.4. Points at extremes

The number of segments should not be too small either. Putting nodes at extremes helps rasterizers to transform glyphs accurately. In particular, repositioning nodes in a slanted font may prove beneficial (Fig. 4D). This implies that a single segment of a Bézier curve should not turn by more than 90 degrees. The points of inflection should also be avoided.

6. A glance at tools for assembling outlines

The set of tools for generating outlines should facilitate complying with the rules summarized in the previous section. We describe here a few representative tools we implemented. Hopefully, it should suffice to convey the most important issues.

6.1. Joining paths

The simplest tool is perhaps an operator for joining paths that have ends nearly coincident:

```
def && = amp_amp_ whatever enddef; % a common
                         % postfix-notation trick
tertiarydef p amp_amp_ q = % |length(p)>0|
 (subpath(0,length(p)-1) of p) ..
  controls (postcontrol length(p)-1 of p)
  and (precontrol length(p) of p) ..
enddef;
```

You use this operator like a normal ampersand, e.g.:

```
p && q && cycle
```

The only difference is that if the edge nodes do not *exactly* coincide, you must not use a single ampersand; double ampersand works in both cases: it simply removes the former of two nodes. (Note that if nodes are actually distant, you may obtain weird results.)

In the METAFONT sources of Computer Modern, tiny segments appear regularly whenever two paths intersect, for example in the arrows of *cmsy10* (Fig. 5). Such segments are harmless from the point of view of generating bitmaps. They can, however, cause quite a lot of commotion from the point of view of outline construction: the path shown in Fig. 5 yields strange values of the `turningnumber` function, depending on the actual resolution: -1

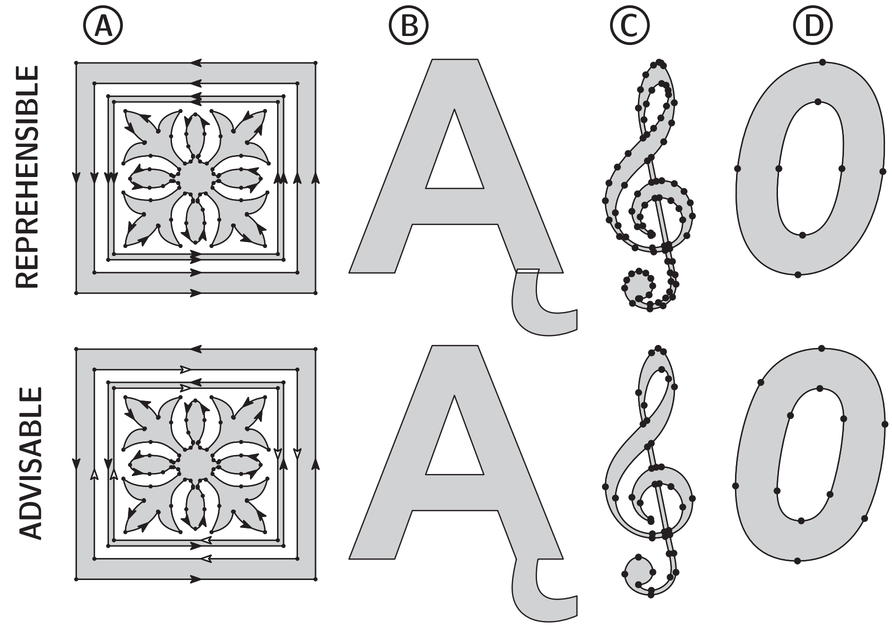

Fig. 4. Common faults of outline design and avoiding them; the letter 'Aogonek' in the top line comes from the standard set of Windows fonts (*Arial CE Bold*).

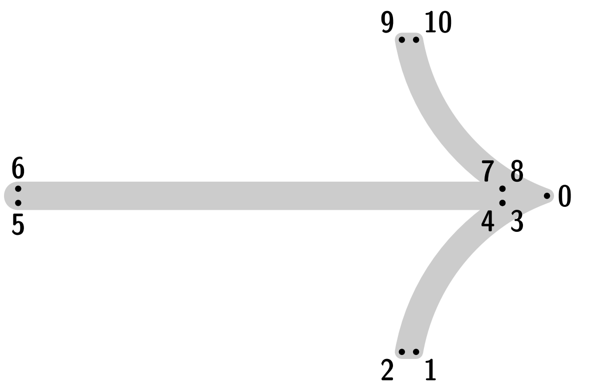

Fig. 5. The rightward arrow from *cmsy10*. The outline of the arrow contains two pairs of nearly coinciding nodes: 3, 4 and 7, 8. Their coordinates for a Linotronic mode, 1270 dpi, read: (150.4547, 42.00003), (150.4547, 41.99976), (150.4547, 46.00024), and (150.4547, 45.99997), respectively, which causes a strange turning number of the path (equal to -3).

for a laser printer mode (300 dpi), and -3 for a phototypesetter mode (1270 dpi).

It is numerical instability that is responsible for the singular behaviour of METAFONT and METAPOST. Tiny segments may form loops having an impact on the result of the turningnumber operation; thus, the important characteristic of a path, its turning number, is not reliable if paths are constructed incautiously.

Tangency is another feature of geometric objects that leads to numerical instability. For example, the turning number for a path defined as

```
(0,0)--(0,-1){up}..(1,1)--cycle
```

is 0, whereas the value becomes 1 after reversing the path. The reason behind this somewhat-amazing result is that the path turns by 180 degrees at the node $(0,-1)$, i.e., the path is locally self-tangent.

In general, the problems caused by numerical instability cannot be solved automatically: only a careful control over the details of the construction of a path may help to overcome the unwanted consequences of the instability. This is exactly the reason why a universal, efficient and reliable converter from a bitmap-oriented METAFONT program into a tidy outline form is hardly imaginable (cf. Sec. Bitmaps versus outlines above).

6.2. A problem in elementary geometry

Elementary geometry prompts constructions that prove useful in computer type design. An example of such a construction is the computation of a side of a right-angled triangle, given its hypotenuse (c) and the length of one of its sides (b):

```
tertiarydef c side b = % |pair c; numeric b;|
 begingroup
  save a; pair a;
  % |(length(c)+-+b)=length(a)|
  c=a+b/(length(c)+-+b)*(a rotated -90);
  a
```

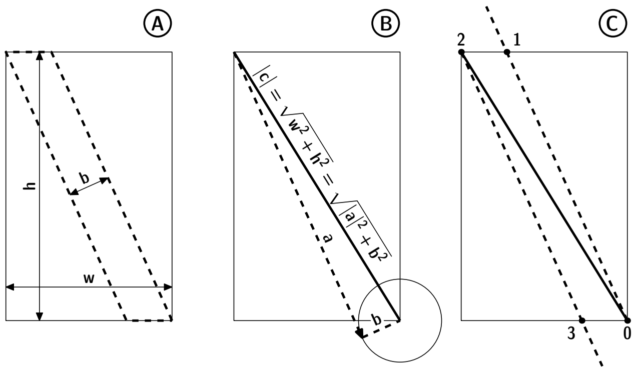

Fig. 6. Inscribing a band (a parallelogram) in a given rectangle: a typical example of construction of a side of a right-angled triangle; more explanations in the text.

```
 endgroup
enddef;
```

The following example illustrates the application of the macro `side`: given the height h and width w of a rectangle, construct a band (parallelogram) of the width b inscribed in the rectangle (Fig. 6A).

The construction with a ruler-and-compasses method is elementary (Fig. 6B):

1. draw the rectangle in question (let's call its diagonal c);
2. draw in one of the corners of the rectangle (say lower-right—node 0 in Fig 6C) a circle of radius b;
3. draw the side a, i.e., a straight line tangential to the circle and passing through the opposite end of the diagonal (node 2).

The crossing point of the side and the base of the rectangle is one of the two remaining corners of the parallelogram (node 3); the other remaining corner (node 1) can be found similarly.

This construction can be translated to METAPOST as follows:

```
z0=(w,0); z2=(0,h);
z1=z0+whatever*(z2-z0 side -b); y1=y2; % this
                                        %fixes node 1
z1-z2=z0-z3; % this fixes node 3 by symmetry
```

The clue is the expression `z0+whatever*(z2-z0 side -b)` which can be interpreted as "somewhere on a straight line covering the side a."

6.3. Slanting: another problem in elementary geometry

Slanting (in the art of type design) is not as simple an operation as it may look at first glance. We already noted that slanting necessitates adding

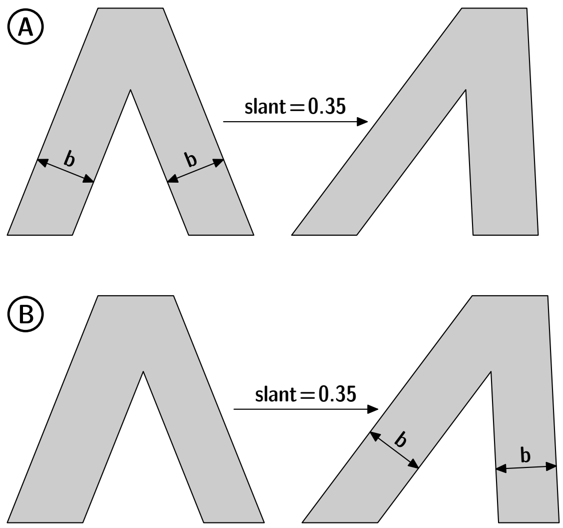

Fig. 7. A capital lambda from an imaginary grotesque font: slanting (exaggerated here) usually affects the width of stems (A), but this effect can (and should) be neutralized (B).

points at extremes (Fig. 4D). Furthermore, slanting should not affect the width of stems (see Fig. 7A).

Assume that the stem slope is given by a vector d and that its resulting breadth after slanting by a slant s should be b; the initial breadth of the stem (i.e., prior to slanting), b', is given by the following formula, in the sequel referred to as the "slant correction formula":

```
b'=b*length(unitvector(d) slanted s)
```

The result of taking this correction into account is displayed in Fig. 7B.

6.4. An advanced problem in elementary geometry

Consider now a "mixture" of two previously demonstrated problems: assume that we want to inscribe a band into a given rectangle in such a way that after slanting by a given slant it will have the width b. Neither the macro 'side' nor the formula for the correction of a stem width can be applied explicitly, as both initial and final slopes depend on each other.

The solution is an iterative algorithm:

1. use the macro 'side' to find the band of the width b for the non-slanted case;
2. find the corrected stem width b' using the slant correction formula;
3. set $b \leftarrow b'$ and repeat the steps (1)–(3) until the process converges (in practice 3–4 steps suffice).

The code is here, in case you're curious:

```
primarydef c /\ b =
% A variant of the |side| procedure that
% iteratively counteracts a slant deformation;
% given: |slant|: a slant (global),
% |c|: the hypotenuse (vector)
% of a right-angled triangle,
% |b|: the length of one of its sides;
% result: the other side of the triangle (vector)
 if slant=0: (c side b) else:
  begingroup save b_,b__, n; b_:=b__:=b; n:=10;
   forever:
    b_:=b*length(unitvector(c side b_)
             slanted slant);
    exitif (abs(b_-b__)<.01) or (n<=0);
    b__:=b_; n:=n-1;
   endfor
   if (abs(b_-b__)>=.01):
     errmessage "Iteration hasn't converged"; fi
   c side b_
  endgroup
 fi
enddef;
```

Examples of the use of this macro can be found in many places in the sources of Antykwa Półtawskiego.

6.5. Removing overlaps

As the last general-purpose tool, we consider the operation of "welding" outlines, also known as "removing overlaps." Most WYSIWYG graphic programs are equipped with such an option. Surprisingly enough, this operation is missing from the set of standard tools for METAFONT/METAPOST.

The set of macros implementing the relevant outline operations, ROEX ([8], [9], [10]), has never become popular (we know about a few people using it) which is understandable, considering the size of the code (70 kB) and the relatively low efficiency and robustness of the employed algorithms.

It should be emphasized, however, that algorithms accomplishing operations of this kind cannot be fully reliable because of the above-mentioned numerical instability problems (cf. Sec. Joining paths above). Nonetheless, when used in a disciplined way, they prove to be extraordinarily useful: the source code is compact and comprehensible. Therefore, for the purposes of work on Antykwa Półtawskiego, we devised a simplified version (4 kB) of the algorithm accomplishing set-theoretic operations (Fig. 8).

The relevant macro takes two cyclic path as arguments, intersects them and assembles the result according to the orientation of the input paths: if both paths are anticlockwise, the result is the union of the two paths, if both are clockwise—the result is the intersection, otherwise the clockwise path is subtracted from the anticlockwise one. In fonts, the first situation occurs most frequently ('x' or 'lslash'), but sometimes, due to the paradigm of having two input paths only, other operations can also be necessary ('oslash').

7. Specific tools for Antykwa Półtawskiego

So far, we have demonstrated a set of basic tools expediting the work on type shapes. Obviously, each font has its own peculiar yet regular features, having an impact on the font appearance as a whole. The glyph programs should reflect all regularities of the font.

7.1. A "building blocks" paradigm

Our general aim was to create a set of "building blocks"—macros returning paths rather than a set of relations (dependencies) such as, e.g., the plain macro penpos returns—and a set of operators for handling the paths. The glyphs are assembled from the "building blocks" with the help of the operators.

Let's look into the code of the letter 'H' (the numbering of lines, of course, does not belong to the code):

```
 1. beginglyph(H);
 2.   save serif_tl, serif_bl, serif_tr,
          serif_br; % local
 3.   path serif_tl, serif_bl, serif_tr,
          serif_br; path letter_H;
 4.  % fix serifs:
 5.   sym_serif4'((wd.H,0), down, uc_stem,
          uc_serif_jut)(serif_br);
 6.   sym_serif3'((wd.H,uc_height), up, uc_stem,
          uc_serif_jut)(serif_tr);
 7.   sym_serif4'((0,uc_height), up,
          uc_stem, uc_serif_jut)(serif_tl);
 8.   sym_serif3'((0,0), down,
          uc_stem, uc_serif_jut)(serif_bl);
 9.  % fix bar:
10.   x1=x4=serif_tl.first.x;
          x2=x3=serif_br.first.x;
11.   y1=y2=1/2uc_height; y4=y3=y2+thin_stem;
12.  % assemble glyph:
13.   letter_H = (serif_br--serif_tr--z3--
       z4--serif_tl--serif_bl--
14.    z1--z2--cycle) start.default;
15.   Fill letter_H;
16.  % final touch:
17.   fix_hstem(thin_stem)(letter_H);
18.   fix_vstem(uc_stem)(letter_H);
19.   fix_hsbw(wd.H,marg,marg);
```

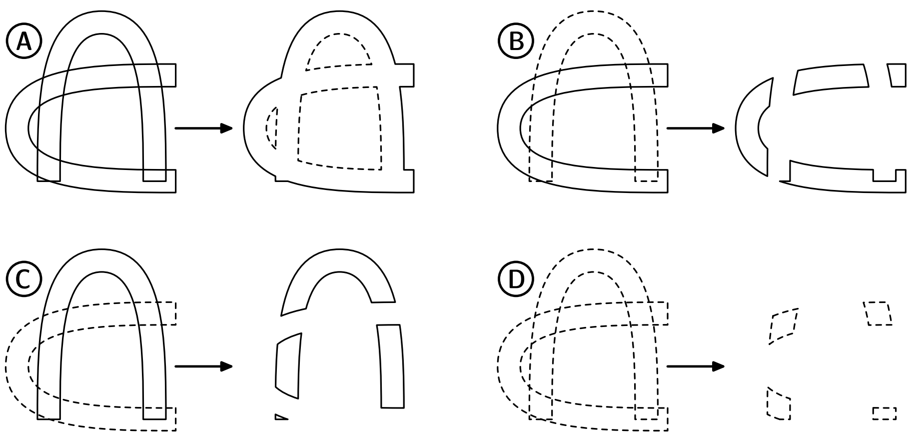

Fig. 8. Rudimentary set-theoretic operations on outlines: union (A), subtraction (C, D), and intersection (D); the operations are accomplished by a single macro, depending on the orientation of paths: anticlockwise paths are marked with a solid line, clockwise with a dashed line.

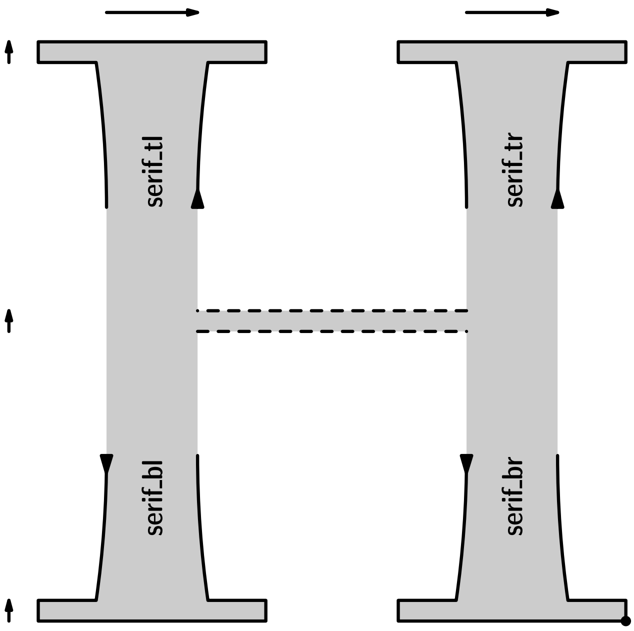

Fig. 9. The letter 'H' from Antykwa Półtawskiego; the construction of letterforms is based on assembling "building blocks," in this case serifs (marked with solid lines); triangles denote the starting points of the serifs, the origin of the whole outline is marked with a circle; arrows show so-called hints.

```
20.  endglyph;
```

The resulting letterform is displayed in Fig. 9.

As you can see, there are several parameters involved in the code: `wd.H` (lines 5, 6, and 19), `uc_stem` (lines 5–8, and 18), `uc_serif_jut` (lines 5–8), `uc_height` (lines 6, 7 and 11), `thin_stem` (lines 11 and 17), and `marg` (line 9). Their meaning is supposed to be self-explanatory.

The macro `sym_serif` (lines 5–8) constructs a serif based on a small number of parameters: the position of a chosen construction point and direction and the sizes of the stem and jut. The result (an open path) is assigned to the last parameter (suffix). The number of the chosen construction node is passed as an optional suffix parameter (see Fig 10 and Sec. Serifs below); in this example, the positions of nodes `3'` and `4'` (lines 6, 8 and 5, 7, respectively) are being fixed.

Such a scheme of defining "building blocks" and assembling outlines out of them has proved very convenient. There are several macros in the Antykwa Półtawskiego package defined in this way. Note, however, that the bar of the letter 'H' is constructed explicitly (lines 10 and 11). It should be emphasized that it is not at all obvious which elements of a font should be classified as "building blocks" and which not (cf. Sec. Unique features below).

Postfix notation also proved to be handy in use in some cases. There are two examples of postfix operators in the quoted code: `first.x` (line 10) and `start.default` (line 14). The former operator returns the x-coordinate of the last node of the argument path (`serif_tl`); the latter operator shifts cyclically the numbering of nodes in such a way that the rightmost lower point of the argument path becomes the first point of the resulting path. We will not dwell too much on this subject, as postfix notation should be regarded as "syntactic sugar" rather than as an important programming technique.

The final portion of the code of the letter 'H' contains three lines that deserve attention:

- Lines 17–18 specify the information about vertical and horizontal lines (stems) which have thickness to be kept uniform throughout the whole font. This information is used by PostScript interpreters for improving the process of conversion of outlines into pixels. The relevant PostScript operators controlling the process of conversion are called hints. Some programs detect hints "automagically" by finding heuristically all feasible distances. We decided that distances could be supplied explicitly without trouble, and thus only the location of hints remains to be found by a heuristic algorithm.
- Line 19 shifts the resulting glyph appropriately in the field of the character, adding left and right side bars (margins; both equal to `marg` in this case). This procedure can be regarded as a simplified version of the `adjust_fit` macro of the Computer Modern programs.

7.2. Serifs

The shape of serifs is the feature of fonts acknowledged as most characteristic. The shapes of serifs of Antykwa Półtawskiego noticeably differ from those of Computer Modern ones; the manner in which the serifs are defined and used is also different.

Let's have a look into the code of the `sym_serif` macro, the heart of the program for the letter 'H.' Actually, it is an instance of a macro defining a serif with non-identical left and right juts, `asym_serif`:

```
 1. vardef sym_serif@#(expr start, in_dir,
          stem, jut)(suffix result) =
 2.  asym_serif@#(start, in_dir, stem,
          jut, jut)(result);
 3. enddef;
 4.
 5. vardef asym_serif@#(expr start, in_dir,
          stem, l_jut, r_jut)
 6.  (suffix result) =
 7. % |@#| -- if not empty, defines which point
 8. % of a serif is to be placed at |start|;
    % otherwise |z.basic| is meant
 9.  clearxy; save v_; path result;
        numeric result.ht, result.wd;
10.  v_=signum(ypart(in_dir));
11.  if (str @# =""): z.basic else: z@# fi=start;
12. % fix central part:
13.  x1-x1'=x6'-x6=v_*spread_wd;
14.  x2-x1=if v_>0: r_jut else: -l_jut fi;
15.  x6-x5=if v_>0: l_jut else: -r_jut fi;
16.  x2-x3=x5-x4=v_*serif_slab*slant;
17.  y.basic'=y0=y7=y2-v_*spread_ht;
18.  y2=y1=y1'=y5=y6=y6'=y.basic-v_*serif_slab;
          y3=y4=y.basic;
19. % fix ending points:
20.  z.basic^in_dir=z.basic';
21.  z0^in_dir=z1'^in_dir=
22.   (z.basic'+1/2stem*stem_corr(in_dir)*
          unitvector(in_dir rotated -90));
23.  z7^in_dir=z6'^in_dir=
24.   (z.basic'+1/2stem*stem_corr(in_dir)*
          unitvector(in_dir rotated 90));
25. % define ''hook'' (alignment) points:
26.  z3'=(x2,y3); z4'=(x5,y4); % for |slant=0|,
                          % |z3=z3'| and |z4=z4'|
27. % complete the construction:
28.  result:=z0{in_dir}..z1--z2--z3--
                z4--z5--z6..{-in_dir}z7;
29.  result.wd:=v_*(x3-x4); result.ht:=v_*(y3-y0);
30. enddef;
```

Although the code looks entangled at first glance, it becomes elementary when illustrated (Fig. 10). Still, it is worthwhile to supply a few words of explanation.

- Line 11: the optional suffix parameter `@#` determines the position of a particular construction point: an empty suffix refers implicitly to `z.basic`; otherwise it is a suffix of any point used in the construction of a serif. Note that all construction points are used locally (`clearxy` in line 9).
- Lines 13 and 16–18: the global parameters used in the macro, namely, `spread_wd`, `spread_ht`, `serif_slab` and `slant`, are common to the upper- and lowercase letters, therefore they do not appear among arguments of the macro.
- Line 16: this line reflects a somewhat uncommon design decision—we assumed that the sides of the serifs should be vertical after slanting, hence an appropriate correction to the position of nodes 3 and 4 is computed. For this reason, nodes $3'$ and $4'$ should be used for vertical alignment prior to slanting.
- Lines 20, 21, and 22: the binary operation `a^b`, defined as `a+whatever*b`, is used in the Antykwa Półtawskiego programs instead of the phrase `whatever[a,b]`; the latter form can be used if both `a` and `b` are known, while the former requires only that the value of `b` is known.

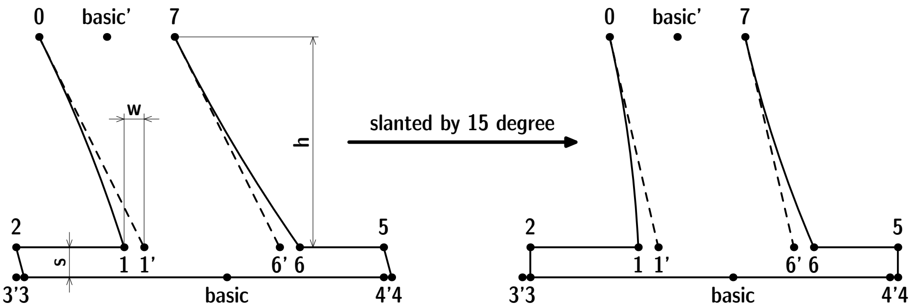

Fig. 10. The construction of the serif of Antykwa Półtawskiego; the distances s, w, and h correspond to the global parameters `serif_slab`, `spread_wd`, and `spread_ht`, respectively. Observe the vertical edges of the serif *after* slanting.

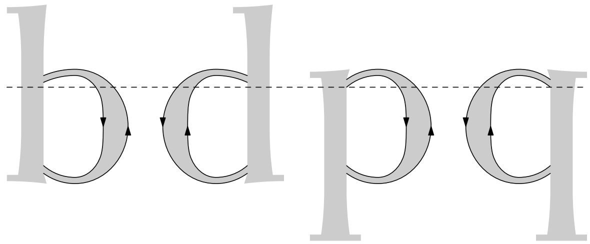

Fig. 11. The construction of the lobes of Antykwa Półtawskiego (marked with a solid line). Observe a slight difference of the lobes in 'b' and 'd' as compared to the lobes in 'p' and 'q' (the drop of the upper part). The job is done by the macro `lc_lobes` which calculates both inner and outer edge appropriately oriented.

- Lines 22 and 24: the `stem_corr` operation computes the slant correction formula for a given slope (`in_dir` in this case; cf. Sec. Slanting: another problem in elementary geometry above).

There exists a variant of a serif used in lowercase letters (see Fig. 11). It is constructed and used in a very similar way to the serif just described, therefore we will skip the code.

7.3. Lobes

Lobes occurring in the lowercase letters 'b,' 'd,' 'p,' and 'q' are essential for the appearance of Antykwa Półtawskiego. In particular, it is important that they are similar, although not necessarily identical: 'b' and 'd' have (probably) different lobes from that of 'p' and 'q' (Fig. 11).

The definition of lobes complies with the scheme that the programs for the construction of serifs manifest. In particular, the header of the macro `lc_lobes` resembles that of `asym_serif`:

```
vardef lc_lobes@#(expr start, width, height,
   raise, drop)(suffix result)
```

There are more kinds of lobes (letters: 'j,' 'J,' 'm,' 'n,' 'u', 'U', etc.), but all of them can be defined using the same paradigm, which unifies and thus facilitates the task of assembling glyphs.

7.4. Unique features

Like the majority of fonts, Antykwa Półtawskiego contains many characteristic details (such as the shape of the bottom part of the letter 'g'—Fig. 3) appearing in only one glyph. Sometimes it is natural to consider them as an intrinsic part of the program for a given glyph, and sometimes it is reasonable to parameterize such local features by defining macros or constants for a single use.

Working on Computer Modern, Knuth apparently faced the same problem. Dozens of coefficients evidently introduced *ad hoc* can be encountered in the Computer Modern programs. For example, the program for the roman letter 'a' contains several "magic" numbers (the most striking ones are underlined):

```
...
if serifs: pos1(flare,180); pos2(hair,180);
pos3(vair,90);
lft x1r=hround max(u,2.1u-.5flare); x3=.5w-.5u;
y1=min(bh+.5flare+2vair+2,.9[bh,h]-.5flare);
bulb(3,2,1);  % bulb
else: pos1(5/7[vair,flare],95);
x1l=good.x 1.5u; x1r:=good.x x1r;
pos3(1/8[vair,thin_join],90);
x3=.5w-.2u; top y1r=vround .82[bh,top y3r];
filldraw stroke term.e(3,1,left,.9,4);
fi  % terminal
pos4(stem,0); rt x4r=hround(w-2.5u+.5stem);
y4=1/3[bh,h];
pos5(stem,0); x5=x4; y5=max(.55bh,2vair);
...
```

It seems that there is no simple answer to the question "to parameterize or not to parameterize?"—the choice is unavoidably controversial.

8. Documenting Antykwa Półtawskiego

Knuth apparently did not think about literate programming *in* METAFONT (although METAFONT itself is written in WEB), therefore there are neither *tangle* nor *weave* equivalents for the METAFONT language. Knuth equipped, however, the METAFONT system with a simple formatting program, MFT, that converts a METAFONT source into a formatted TEX document. MFT is smart enough to be used in a *literate* style, though obviously only for "weaving."

The basic idea is the same as in Knuth's WEB system: the METAFONT source is interleaved with TEX code. It can be achieved due to a peculiar MFT convention, in that a double percent sign has dual meanings: for METAFONT it denotes just a usual comment, whereas for the MFT program it means that the text appearing after a double percent is to be inserted verbatim into the resulting TEX document.

Obviously, the MFT program can also be used for formatting METAPOST sources. A typical fragment of the "literate documentation" of Antykwa Półtawskiego looks as follows:

```
%% \--------------------------------------------
%% In general, all objects are supposed to be
%% drawn by the {\bf endglyph} macro, i.e., all
%% drawing operations are deferred. The same
%% concerns labelling, which necessitates
%% redefinition of labelling macros.
%% \-
%% Zak/lada si/e, /ze wszelkie operacje rysowania
%% s/a  ,,odraczane'' i~realizowane dopiero przez
%% makro {\bf endglyph}. To samo dotyczy
%% etykietowania, sk/ad konieczno/s/c
%% przedefiniowania makr etykietuj/acych.
%% \---------------------------------------------
vardef pen_labels@#(text t) =
 if project>2: % proofing level
  forsuffixes $$=l,,r: forsuffixes $=t:
   if known z$.$$: makelabel@#(str$.$$,z$.$$) fi;
  endfor endfor
 fi
enddef;
```

In this case, the TEX code invokes the macro \- which typesets the bilingual description of the METAPOST source. The ensuing METAFONT code is formatted by the MFT program. The result of the typesetting process is displayed in Fig. 12.

METAPOST, like METAFONT, is capable of generating hardcopy proofs. Actually, METAPOST proofs are EPS files, therefore they can be used in a wider context than METAFONT ones. In particular, they can be included into the formatted sources. We used this technique in the Antykwa Półtawskiego sources. The examples of pages containing proof illustrations are shown in Fig 13.

9. Postprocessing METAPOST output

The idea of postprocessing is rooted in the design principles of TEX. There are several TEX-oriented utilities belonging to the standard TEX distribution: POOLTYPE, DVITYPE, TFTOPL, PLTOTF, VFTOVP, and VPTOVF. In a sense, all of them, as well as TEX drivers, can be regarded as postprocessors.

There are also METAFONT-oriented postprocessors: MFT, GFTYPE, GFTOPK, and PKTYPE (TFTOPL and PLTOTF could also be included here).

So far, there are no publicly released postprocessors for METAPOST, but sooner or later they are bound to appear, the more so as the METAPOST system is, actually, based on postprocessing (METAPOST → TEX → DVI → METAPOST).

Knuth insisted that his WEB system was independent of external systems: he programmed all the above-mentioned utilities in PASCAL. In the early 'eighties such an attitude was plausible, but—in our humble opinion—it no longer is. Nowadays, there exist stable, public domain systems for processing text and binary files. Our experience shows that a lot can be achieved with the aid of a simple text file processor, AWK ([11]). A few years ago, it was adopted as a part of the Gnu Project (GAWK) which guarantees both stability and maintenance in the future. Since METAPOST generates well-formed 7-bit ASCII files, we decided to use GAWK for postprocessing METAPOST output.

A *Type 1* font consists of a binary file PFB (or its hexadecimal equivalent, PFA), containing the description of glyph shapes, and should be accompanied by a text metric file AFM.

In theory, GAWK might have written the hexadecimally encoded PFA files, but there is another tool, better suited for this purpose, namely, the T1UTILS package ([13]). It contains both an assembler and a disassembler of *Type 1* fonts. Obviously, we need the assembler first of all, but the disassembler also proves useful from time to time.

Having METAPOST, GAWK and the *Type 1* assembler, we are ready to accomplish the process of generating Type 1 fonts. It consists of three steps:

In general, all objects are supposed to be drawn by the **endglyph** macro, i.e., all drawing operations are deferred. The same concerns labelling, which necessitates the redefinition of labelling macros.	Zakłada się, że wszelkie operacje rysowania są „odraczane" i realizowane dopiero przez makro **endglyph**. To samo dotyczy etykietowania, skąd konieczność przedefiniowania makr etykietujących.

vardef pen_labels @#(**text** t) =
  if *project* > 2: `% proofing level`
    forsuffixes $\$\$ = l, , r$: **forsuffixes** $\$ = t$:
      if known $z_{\$.\$\$}$: **makelabel** @#(**str** $\$_{\$\$}$, $z_{\$.\$\$}$) **fi**;
    endfor endfor
  fi
enddef;

Fig. 12. An example of the formatting of the documentation of Antykwa Półtawskiego: the excerpt from the documentation corresponding to the source quoted above.

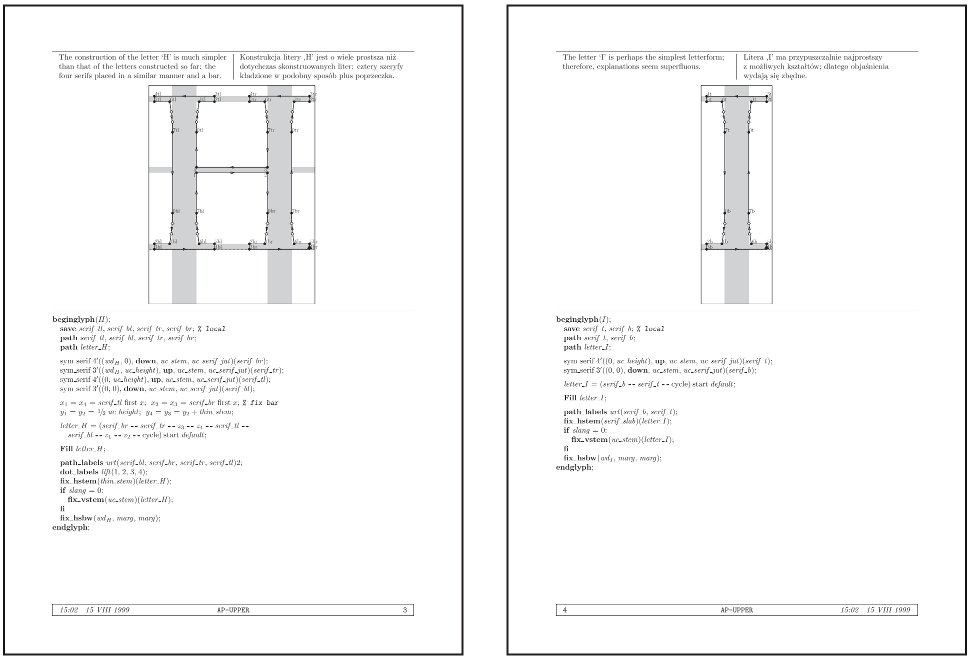

Fig. 13. The appearance of the documentation of Antykwa Półtawskiego: the METAFONT source can contain references to proofs.

1. METAPOST compiles the font source and writes each glyph to a separate EPS file; additional information (e.g., hints, the name of a glyph, etc.) is written by `special` commands and stored as PostScript comments after the header of an EPS file.

2. GAWK processes the resulting EPS files along with a few auxiliary (configuration) files and creates an AFM file and an input file for the *Type 1* assembler.

3. The *Type 1* assembler translates the intermediate result generated by GAWK into a PFB file.

In order to use the resulting font in TEX, one more step is needed: the installation of a font.

The main part of the task of the installation of a PostScript font is generating the relevant TEX font metric file—TFM. This can be performed by using, e.g., a popular program AFM2TFM by Tomas Rokicki. There are several packages facilitating the installation of PostScript fonts for TEX, most notably *fontinst* ([12]). We decided, however, not to include any of them into our METAPOST-to-*Type 1* machinery.

The best tool for generating TFM files is of course METAFONT. At present, the sources of

Antykwa Półtawskiego are unuseable by METAFONT. We are thinking about preparing a format acceptable to both METAFONT and METAPOST and adding one more step to the process of the generation of *Type 1* fonts: generating a TFM file by METAFONT from the same source. It seems that, in general, there are no fundamental obstacles, but, as wise people say, don't count your chickens before they are hatched.

10. Conclusions

The process of generating *Type 1* fonts turned out to be fairly efficient. It is comparable with the process of generating PK files. Therefore, it is possible to generate the font variations on the fly, in accordance with our postulates (see Sec. Antykwa Półtawskiego). There remains the problem of the TEX-METAPOST interface, but that must be a matter for the future.

There is, however, a distressing side of the story: is it worthwile to put a lot of energy into an obsolescent *Type 1* format? What about *Multiple Master*, *TrueType*, *OpenType*, and a multitude of other fonts?

Adobe PostScript is a stable worldwide standard. So far, Adobe have paid attention to the backward compatibility of PostScript versions. We hope that the situation will endure for a few years more and that *Type 1* fonts will be usable for a pretty long period.

From the point of view of TEX applications, the quality and the functionality of *Type 1* fonts is sufficient. The advantages that can be gained by using METAPOST seem at least equivalent, if not more valuable, than the glorified features offered by the above-mentioned font formats. For example, you may regard the technique described in this paper as a TEX-oriented implementation (and, in some respects, a generalization) of the most important features of *Multiple Master* fonts.

Knuth says in his The METAFONTbook: It seems clear that further work with METAFONT has the potential of producing typefaces of real beauty. This, nobody can deny. We believe, however, that replacing the word "METAFONT" by "METAPOST" offers even better prospects.

Bibliography

[1] Donald E. Knuth, The METAFONTbook, Addison-Wesley, seventh printing, 1992.

[2] Andreĭ Slepukhin, private communication, January, 1996.

[3] Richard J. Kinch, MetaFog: converting METAFONT shapes to contours, TUGboat **16** (3), pp. 233–243, 1995.

[4] Richard J. Kinch, Belleek: A call for METAFONT revival, Proc. of 19th Annual TUG Meeting, AuGUST 17–20, 1998, Toruń, Poland, pp. 131–136.

[5] Taco Hoekwater, Generating Type 1 fonts from METAFONT sources, Proc. of 19th Annual TUG Meeting, AuGUST 17–20, 1998, Toruń, Poland, pp. 137–147.

[6] John D. Hobby, `http://cm.bell-labs.com/who/hobby/MetaPost.html`

[7] Adobe Type 1 Font Format, Addison-Wesley, 1990.

[8] Bogusław Jackowski and Marek Ryćko, Labyrinth of METAFONT paths in outline, Proc. of 8th European TEX Conference, September 26–30, 1994, Gdańsk, Poland, pp. 18–32.

[9] Bogusław Jackowski, A METAFONT-EPS interface, Proc. of 9th European TEX Conference, September 4–8, 1995, Arnhem, The Netherlands, pp. 257–271.

[10] ROEX: a METAFONT macro package accomplishing operations on paths, commonly known as "removing overlaps" and "expanding strokes," `ftp://ftp.GUST.org.pl/TeX/graphics/MF-PS/roex/`.

[11] Alfred V. Aho, Brian W. Kernighan, Peter J. Weinberger, The AWK Programming Language, Addison-Wesley, 1988.

[12] Alan Jeffrey, Rowland McDonnell, Ulrik Vieth, Fontinst: the font installation package, `ftp://ftp.gust.org.pl/TeX/fonts/utilities/fontinst/`.

[13] I. Lee Hetherington, Utilities for assembling and disassembling Type 1 fonts, `ftp://ftp.gust.org.pl/TeX/fonts/utilities/t1utils.zip`

Acknowledgements

The authors express their hearty thanks to the Polish TEX Users Group GUST for partially sponsoring the work on Antykwa Półtawskiego and to Phil Taylor for many valuable comments concerning the presentation of the work.

◇ Bogusław Jackowski
B.Jackowski@GUST.org.pl

◇ Janusz M. Nowacki
J.Nowacki@gust.org.pl

◇ Piotr Strzelczyk
piotrs@telbank.pl