Zebrackets: A score of years and delimiters

Michael Cohen, Blanca Mancilla and John Plaice

1 Introduction

In this paper, we present the resurrection of the *Zebrackets* project, originally initiated by the first author 20 years ago, with which parentheses and brackets are *zebra-striped* with context information. There are two reasons for this innovation: first, to improve visual presentation of the necessary linearization of hierarchical structures in text, and second, to make a first step away from the assumption that documents must be built up from a set of unchanging atoms called characters.

Parentheses and other pairwise delimiters are important because they are the primary way by which text, which is serialized, can denote higher-order dimensionality. For example, two-dimensional structures can be directly expressed, as in $\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$. For 2D data structures such as matrices, such graphical expression is natural, but unnecessary, as serial expression is logically equivalent, albeit less perspicuous, as in $((a_{11}\ a_{12})\ (a_{21}\ a_{22}))$, and generalizable to arbitrary rank and dimension, as in $(a_1\ (a_{21}\ a_{22})\ (a_{31}\ (a_{321}\ a_{322})))$. Such notation usually assumes "row-major" order, in which the horizontal index changes fastest in a canonical (depth-first) enumeration. This convention can be made explicit, by introducing grouping delimiters, as in $\begin{pmatrix} (a_{11} & a_{12}) \\ (a_{21} & a_{22}) \end{pmatrix}$. The transpose ("column-major") representation, is given by $\left(\begin{pmatrix} a_{11} \\ a_{12} \end{pmatrix} \ \begin{pmatrix} a_{21} \\ a_{22} \end{pmatrix} \right)$.

Zebrackets[1] were originally developed more than a score (20) of years ago, demonstrating the expressive power of microarticulated glyphs [Coh92] [Coh93] [Coh94]. The basic idea is to allow parentheses [Len91] and square brackets to take on stripes and slits ("poles 'n' holes") to carry extended information, such as functional rôle, logical position, and nesting level in an expression. Pairwise delimiters are "scored", by cutting aligned typographical grooves, to associate balanced mates and ease visual parsing.

Below, we show one possible scoring of this sample text:

[(a [b (c) d] [e (f) g]) (a [b (c) d] [e (f) g])]

Here it is, with greatly magnified brackets and parentheses:

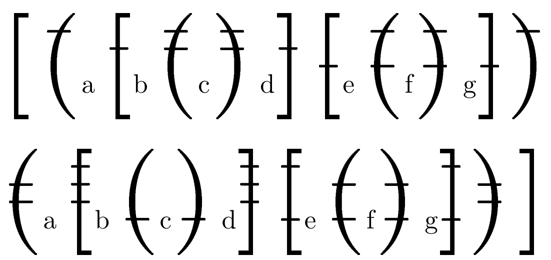


Table 1 shows a number of examples of the use of the Zebrackets infrastructure on the same sample text.

The intervening score of years has not been especially kind to the original implementation: it was hardly sturdier than a "paper-clips and bubble-gum" contraption in the first place, and the slide into deprecation and disuse of METAFONT, accelerated by the emergence of PDF as the interchange format of choice, which cannot natively use characters generated by METAFONT, hastened the obsolescence of the Zebrackets prototype. Adobe's "Multiple Masters" (such as Adobe Sans and Adobe Serif) and Apple's TrueType GX were similarly ahead of their time, and failed to achieve critical mass and widespread adoption. Jacques André's contextual fonts, dynamic fonts [AO89] [AB89], and Scrabble font [And90] were Type 3, so also withered.

Nevertheless, we believe that the principles underlying the system are still valid. There is a huge multidimensional space of potential characters and glyphs, too big to be precompiled, and so a lazy, demand-driven, image-time generation, both of fonts and glyphs, with caching or memoization, as is used in dynamic programming, is the only practicable solution. Contemporary assumptions about fonts do not allow this possibility [Har07], so reviving the existing implementation strategy is still of relevance. The presentation here presents the font structure, and the use, both implicit and explicit, of these fonts.

2 The fonts

The Zebrackets project relies on a set of fonts generated from the METAFONT version of the Computer Modern fonts. The names of the fonts are all of the form `z`$(a)(b)(c)(d)(e)$, where:

(a) is a single letter, either '`b`' or '`p`'; the font contains either all brackets (`b`) or all parentheses (`p`).

(b) is a single letter, one of '`b`', '`f`', or '`h`'; the marks in the font are all either slots (`b` for background), ticks (`f` for foreground), or ticks within slots (`h` for hybrid).

(c) is a single letter, one of '`a`' through '`h`'; the font will contain 2^m pairs of left and right delimiters,

[1] The name, suggested by Bob Alverson, is a play on words as the delimiters resemble zebra stripes: (ze(bra)kets).

Table 1: Stripes, slits, and slots: Examples of zebrackets with various arguments. Each zebracket has a set of slots (here computed automatically), which can be striped according to the chosen style: plain “foreground” stripes (style ‘f’ in the table); more subtle, erasing “background” slits (style ‘b’); or “hybrid” (style ‘h’), which creates a slit for each slot, then places foreground stripes on top thereof. Stripes generation can automatically count unique pairs or track nesting depth, or count unique pairs at a given depth (“breadth”). The encoding can be unary, binary, or “demultiplexing”, up through the maximum as calculated by initial pass of a parser. Note that all encodings have 0 as origin, but the rendered index origin can be changed to unity.

`encoding`	`style`	`index` = unique	`index` = depth	`index` = breadth
unary	b	(a (b (c) d) (e (f) g))	(a (b (c) d) (e (f) g))	(a (b (c) d) (e (f) g))
	f	(a (b (c) d) (e (f) g))	(a (b (c) d) (e (f) g))	(a (b (c) d) (e (f) g))
	h	(a (b (c) d) (e (f) g))	(a (b (c) d) (e (f) g))	(a (b (c) d) (e (f) g))
binary	b	(a (b (c) d) (e (f) g))	(a (b (c) d) (e (f) g))	(a (b (c) d) (e (f) g))
	f	(a (b (c) d) (e (f) g))	(a (b (c) d) (e (f) g))	(a (b (c) d) (e (f) g))
	h	(a (b (c) d) (e (f) g))	(a (b (c) d) (e (f) g))	(a (b (c) d) (e (f) g))
demux	b	(a (b (c) d) (e (f) g))	(a (b (c) d) (e (f) g))	(a (b (c) d) (e (f) g))
	f	(a (b (c) d) (e (f) g))	(a (b (c) d) (e (f) g))	(a (b (c) d) (e (f) g))
	h	(a (b (c) d) (e (f) g))	(a (b (c) d) (e (f) g))	(a (b (c) d) (e (f) g))

Table 2: Font `zphecmr12`.

	`0/8`	`1/9`	`2/A`	`3/B`	`4/C`	`5/D`	`6/E`	`7/F`
`0`	(	(	(	(	(	(	(	(
	(	(	(	(	(	(	(	(
`1`	)	)	)	)	)	)	)	)
	)	)	)	)	)	)	)	)

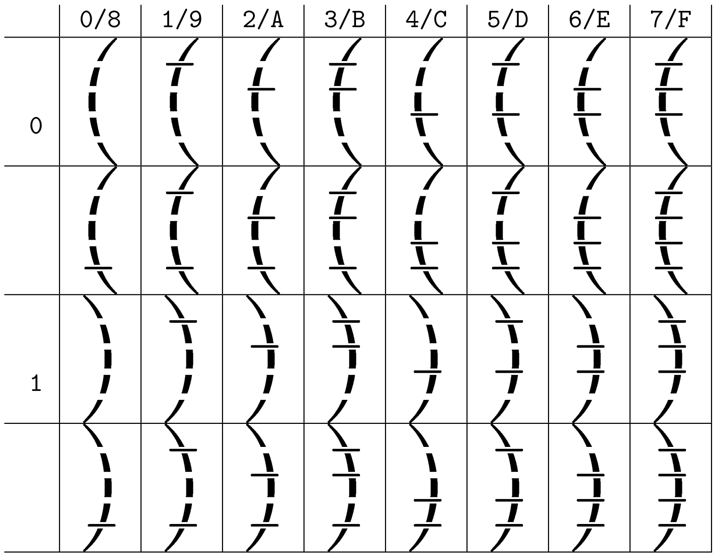
Table 3: Font `zbfdcmtt12`.

	`0/8`	`1/9`	`2/A`	`3/B`	`4/C`	`5/D`	`6/E`	`7/F`
`0`	[	[	[	[	[	[	[	[
	]	]	]	]	]	]	]	]

Table 4: Font `zphecmr12, magnification` 2.

	`0/8`	`1/9`	`2/A`	`3/B`	`4/C`	`5/D`	`6/E`	`7/F`
`0`	(	(	(	(	(	(	(	(
	(	(	(	(	(	(	(	(
`1`	)	)	)	)	)	)	)	)
	)	)	)	)	)	)	)	)

Table 5: Font `zphecmr12, magnification` 4.


0/8
1/9
2/A
3/B
4/C
5/D
6/E
7/F
0
1

with $m \in 0..7$, where `a` corresponds to $m = 0$ (i.e., 1 pair or 2 glyphs), `b` corresponds to $m = 1$ (i.e., 2 pairs or 4 glyphs), …, and `h` corresponds to $m = 7$ (i.e., 128 pairs or 256 glyphs).

(d) is the name of a Computer Modern font family, such as '`cmr`'.

(e) is a font size, such as '`10`'.

We consider two examples. In the first example, font `zphecmr12` (Table 2) was generated by calling the `zebraFont.py` script with arguments specifying parentheses striped in a hybrid visual style across 4 slots, using 12 pt Computer Modern Roman as the base:

```
python3 zebrackets/zebraFont.py
--kind parenthesis --style hybrid
--slots 4 --size 12 --family cmr
```

The font contains exactly $2^{4+1} = 32 =$ `0x20` parentheses. In the first half of the table, for $i \in$ `0x00` .. `0x0F`, glyph i is an opening (left) parenthesis, encoding i as a binary number with ticks placed in four always-drawn slots. Similarly, in the second half of the table, for $i \in$ `0x10` .. `0x1F`, glyph i is a closing (right) parenthesis encoding ($i -$ `0x10`) as a binary number.

In the second example, font `zbfdcmtt12` (Table 3) was generated by:

```
python3 zebrackets/zebraFont.py
--kind bracket --style foreground
--slots 3 --size 12 --family cmtt
```

specifying square brackets striped, using a foreground style, across 3 slots, with the 12 pt typewriter font as a base. The font contains exactly $2^{3+1} = 16 =$ `0x10` brackets. For $i \in$ `0x00` .. `0x07` across the top of the table, glyph i is an opening bracket encoding i as a binary number with ticks placed in three fixed slots. Similarly, in the bottom half of the table, for $i \in$ `0x08` .. `0x0F`, glyph i is a closing bracket encoding ($i -$ `0x08`) as a binary number.

The `zebraFont.py` script generates a new METAFONT file whose name is the font name followed by `.mf`, placing that file in the directory

```
$TEXMFHOME/fonts/source/public/zbtex
```

then calling `mktextfm` on that font name, before finally calling `mktexlsr` to add the generated `.mf`, `.tfm`, and `.600pk` files to the cache for `$TEXMFHOME`.

Thus, for the second example above, the script `zebraFont.py` generates the METAFONT file:

```
.../source/public/zbtex/zbfdcmtt12.mf
```

and `mktextfm` generates files (assuming `ljfour` is the default output mode):

```
.../tfm/public/zbtex/zbfdcmtt12.tfm
.../pk/ljfour/public/zbtex/zbfdcmtt12.600pk
```

and `mktexlsr` updates the cache of the list of files:

```
$TEXMFHOME/ls-R
```

We can also make magnified versions of these fonts. Our third example (Table 4) is the `zphecmr12` font with magnification 2, which corresponds to TeX magnification $\sqrt{2} \approx 1.414$. The font was generated by:

```
python3 zebrackets/zebraFont.py
--kind parenthesis --style foreground
--slots 3 --size 12 --family cmr
--magnification 2
```

Our last example (Table 5) shows the same font with magnification 4, corresponding to TEX magnification 2. The font was generated by:

```
python3 zebrackets/zebraFont.py
--kind parenthesis --style foreground
--slots 3 --size 12 --family cmr
--magnification 4
```

When a magnification argument is passed to `zebraFont.py`, the `mf-nowin` and `gftopk` scripts are called to produce larger versions of the fonts. The magnification argument is multiplied by 600. Hence we get:

```
2  .../zbfdcmtt12.1200pk
4  .../zbfdcmtt12.2400pk
8  .../zbfdcmtt12.4800pk
```

The METAFONT file that is created by the script `zebraFont.py` is an eight-line file. It inputs the base Computer Modern font, sets the parameters for the number of slots and whether the marks are foreground, background, or hybrid, then inputs a file for generating a set of parentheses or a set of brackets. For example, font `zpfbcmr10.mf` contains the following lines:

```
if unknown cmbase: input cmbase fi
mode_setup;
def generate suffix t = enddef;
input cmr10; font_setup;
let iff = always_iff;
stripes:=1;
foreground:=1;
input zeromanp;
```

The last file input, by the last line above, is `zeromanp.mf`, which is derived from the original Computer Modern `roman.mf` (prefixing "ze" to indicate its adaptation to zebrackets). It is one of four METAFONT files distributed with the *Zebrackets* project. The `zeromanp.mf` file sets some parameters, then inputs the punctuation file `zepunctp.mf`, itself derived from the Computer Modern `punct.mf`. For brackets, there are corresponding files `zeromanb.mf` and `zepunctb.mf`.

3 Using the fonts explicitly

There are two ways to use the fonts generated by the *Zebrackets* project: explicitly and implicitly. In this section, we present the explicit approach, and show how it is used to produce the bibliographic references of this article.

Suppose that a font `zbfhcmr10` has been generated and we wish to use it. Then we need to declare the font and its size with a line like this — the `J` encodes size 10 and the `A` encodes magnification 1:

```
\ifundefined{zbfhcmrJA}
\newfont{\zbfhcmrJA}
{zbfhcmr10 scaled 1000}\fi
```

Font `zbfhcmr10` is a font of 256 brackets, all with foreground ticks. To [bracket some text] with a pair of delimiters with four selected ticks (in this example, using all but two slots at the top and one at the bottom), the `.tex` source code can use

```
{\zbfhcmrJA\symbol{60}} bracket some
       text {\zbfhcmrJA\symbol{188}}
```

since binary $0111100 = 2^5 + 2^4 + 2^3 + 2^2 = 60$ and $128 + 60 = 188$.

To hint at the expressive flexibility of such functionality, the bibliographic references of the original article on Zebrackets [Coh93] placed a tick for each page in which a `\cite`-ation appeared in the left bracket of the corresponding citation label and a tick for each page in which a `\nocite`-ation appeared in the right bracket.

In this article, we show the same extension throughout (not just in the bibliographic References section), with the now explicit convention that should a document have more than seven pages, then all references beyond the seventh page activate the seventh tick. In the reference [Coh93], the activation of the first and fifth ticks in the opening bracket indicate that reference's citation on the 1st page of this paper as well as here on the 4th page.

Below is some of the code needed for this functionality. There are two counters used as temporary variables:

```
\newcounter{bracei}
\newcounter{bracej}
```

For each citation x, a pair of counters is set up, `ze:`x for the left bracket, and `zeno:`x for the right bracket. The `\zecite` macro is like the standard LATEX `\cite` macro, but it also calls `\zecitation`, which bitwise-ors-in 2^{p-1} to counter `ze:`x for the left bracket, should there be a `\zecite{`x`}` on page number p:

```
\newcommand{\zecite}[2][]%
{\def\tmp{#1}\ifx\tmp\@empty\cite{#2}%
             \else\cite[#1]{#2}\fi
 \zecitation{#2}}

\newcommand{\zecitation}[1]%
{\ifundefined{c@ze:#1}%
 \newcounter{ze:#1}%
 \setcounter{ze:#1}{0}%
 \newcounter{zeno:#1}%
 \setcounter{zeno:#1}{128}\fi
 ...
 \addtocounter{ze:#1}{...}}
```

There are macros corresponding to `\nocite`, namely `\zenocite` and `\zenocitation`, bitwise-oring-in 2^{p-1} to counter `zeno:`x for the right bracket, should there be a `\zenocite{`x`}` on page number p.

Finally, the generation of the `\bibitem` is extended, so that

```
{\zbfhcmrJA\symbol{\arabic{ze:x}}}
```

appears as the citation's left bracket, and

```
{\zbfhcmrJA\symbol{\arabic{zeno:x}}}
```

appears as its right bracket.

4 Using the fonts implicitly

Although providing an explicit interface to the Zebrackets infrastructure provides great flexibility, most of the time such invocation is "under the hood" and used implicitly, through the use of pseudo-LaTeX commands appearing in a LaTeX document.

A Zebrackets-enabled LaTeX file (with conventional extension `.zbtex`) is passed through a preprocessor, `zebraParser.py`, which recognizes four constructs:

1. `\zebracketsfont` declares the need for a font, provoking its creation should it not exist.
2. `\zebracketsdefaults` sets default values for the parameters of the other two constructs.
3. `\zebrackets text` designates text in which the parentheses and brackets are to be replaced automatically with zebrackets (including "zeparentheses").
4. `\begin{zebrackets}` ··· `\end{zebrackets}` designates a block of text for the same treatment as for `\zebracketstext`.

Because of this precompilation, from `.zbtex` to `.tex`, the workflow for such zebracketed word-smithing is not as convenient as with, for instance, TeXShop:[2] Compilation can be managed in Unix-like shells with a `Makefile` to check dependencies and invoke the required processes, but there is no automatic preview, synchronization, or other accustomed conveniences.

4.1 The `\zebracketsfont` instruction

The previous section explained how Zebrackets fonts are generated by the `zebraFont.py` script. This script cannot be called directly from a LaTeX document, but can be invoked indirectly through the `\zebracketsfont` instruction. Consider, for example, the following call to `zebraFont.py`:

```
python3 zebrackets/zebraFont.py
--kind parenthesis --style foreground
--slots 7 --size 10 --family cmr
--magnification 1
```

[2] `http://pages.uoregon.edu/koch/texshop/`

The invocation of that call can be made implicitly in the LaTeX document with the following line.

```
\zebracketsfont[
  kind=parenthesis,style=foreground,
  slots=7,size=10,family=cmr,
  magnification=1]
```

As a prelude to LaTeX compilation, the preprocessing script `zebraParser.py` reads and parses this line, directly calls `zebraFont.py` with the appropriate parameters, and removes the line from the LaTeX document, which is exported with the usual `.tex` file extension.

One need not include the full set of key–value arguments, as default values can be used (as explained below). Further, each of the parameter names can be abbreviated, down to just the first three letters, and the keyword arguments can also be abbreviated, as in:

```
\zebracketsfont[kin=p,sty=f,slo=7,
                siz=10,fam=cmr,mag=1]
```

The `\zebracketsfont` instruction takes six arguments, which can appear in any order:

1. `kind` can be either `parenthesis` (`p`) or `bracket` (`b`).
2. `style` can be any one of `foreground` (`f`), `background` (`b`), or `hybrid` (`h`).
3. `slots` is a natural number between `0` and `7`, inclusive.
4. `size` is a natural number for a font size, such as `10` or `12`.
5. `family` is a Computer Modern font family name, such as `cmr` or `cmtt`.
6. `magnification` is a natural number between `1` and `32`, inclusive, representing the square of the TeX font magnification, i.e., a power of $\sqrt{2}$.

4.2 The `\zebracketsdefault` instruction

If Zebrackets is used extensively within a document, then a lot of calls thereto are made, perhaps with similar or even identical parameters. In order to reduce typing (and introduction of errors), default values for any of the Zebrackets parameter names can be assigned.

For example, in the following lines, four fonts are declared, all of family `cmr`, size `10`. All but one use parentheses, all but one are foreground style, and all but one have seven slots.

```
\zebracketsdefaults
  [size=10,family=cmr,
   slots=7,kind=parenthesis,
   style=foreground]
```

```
\zebracketsfont[]
\zebracketsfont[kind=bracket]
\zebracketsfont[style=background]
\zebracketsfont[slots=1]
```

4.3 The `zebrackets` environment

When `zebraParser.py` is called, whenever it parses text to be transformed (when the document contains either the `\zebracketstext` command or the `zebrackets` pseudo-LaTeX environment), then the `zebraFilter.py` script is called. The latter reads the text, determines what fonts are needed (invoking `zebraFont.py`, as necessary), then replaces the brackets and parentheses in the text with font–symbol pair invocations.

Consider the following example, presented in §3 with explicit font–symbol pairs:

[bracket some text]

That example can also be generated implicitly, with the lines:

```
\begin{zebrackets}
  [style=f,number=60,
   slots=7,encoding=binary]
[ bracket
some text ]
\end{zebrackets}
```

The `number=60,slots=7` specifications in the parameter list summons a font using seven slots, from which glyph `60` ($= 2^5 + 2^4 + 2^3 + 2^2$) and its partner `188` ($= 2^7 + 60$) are drawn.

For automatic processing, inputs are handled as follows:

- Parameter `index` can take one of three possible values — `unique`, `depth`, `breadth` — as exemplified in Table 1.
- Parameter `number` overrides the settings for parameter `index`. When `number` is set, all parentheses and brackets in the text being processed get that specific glyph in the font.
- When a value for parameter `slots` is not provided, then the number of slots for the fonts is the minimum needed in order to encode all of the glyphs for the text (taking into account the value of the index parameter).

For example,

(a (b (c) d) (e (f) g))

was generated by the lines:

```
\begin{zebrackets}
  [index=depth,enc=unary,style=f]
(a (b (c) d) (e (f) g))
\end{zebrackets}
```

There are also three additional parameter pairs, each with two values:

- `mixcount=true` states there should be a single counter for striping parentheses and square brackets; `mixcount=false`, two distinct counters.
- `origin=0` states that counting starts from zero; `origin=1`, from one.
- `direction=topdown` means that striping starts from the top of delimiters, whereas `direction=bottomup` starts from the bottom.

The automatic striping of delimiters in a region of text is done with a two-pass algorithm: a) the maximal depth and breadth, and the number of distinct delimiter pairs are computed, in order to determine the number of distinct slots needed (maximum of 7), and b) the correct fonts are generated, if need be, and the correct LaTeX source is created.

5 Conclusion

The Zebrackets infrastructure does not assume that characters are changeless atoms, as standard computing infrastructures do. We consider below this innovation from several perspectives.

5.1 Representative characters

The idea of characters or words as pictures is of course not new. Most characters — including Chinese characters, Japanese kana, and the Roman alphabet — have origins in pictographic associations, albeit with prehistoric abbreviations and stylizations that make the original inspiration obscure or all but indiscernible. Illuminated manuscripts often embellished initials with vines, flowers, animals, and other inventions. Almost a century ago, Apollinaire published books [dK25] featuring “calligrams”, instances of “concrete poetry” or “visual poetry”, in which the typeface and arrangement of words on a page informs the meaning of a poem as much as the words themselves. Contemporary typography often plays with pictorial suggestions [KH08], especially for special-purpose or display faces.

5.2 Context sensitivity

TeX has always featured non-locality, including “butterfly-effect” propagation, in which, for instance, a seemingly small change at the end of a document can affect layout at the beginning, especially in the presence of floating figures and tables. However, such effects are large-scale, macroscopic, rearranging the glyphs, but not mutating the glyphs themselves. Zebrackets suggests subatomic alteration, analog isotopes of the heretofore inviolate characters. A character is the smallest visual part of a notational system that has semantic value. A glyph is one possible representation of a character. Ligatures can be thought of as locally context-sensitive glyph adaptation, as can some kinds of accents, kerning, and hyphenation. But Zebrackets represents a larger context sensitivity, adapting symbols to the broader circumstances. In an extreme case, its filters could be applied to an entire document.

5.3 Analog articulation

Fonts can be thought of as embeddable in a manifold[3] [CK14], and perturbations on this manifold are equivalent to variations of the font characteristics. Microtypography [Kar15] is an unexploited aspect of font design and electronic publishing. Zebrackets challenges the assumption that a glyph is the smallest representation of a character that has semantic value. Such capability hints at giving glyphs depth, not in the sense of a 3D, sculptural sense [Ann74] [FVJ11] [HF13], but logical depth, in the sense of alternate projections of a set of variations on a character. Current technology discourages such generality, and, since the character/glyph/font system is so deeply and tightly interwoven with any operating system, application, or program, traditional computer typography and character-handling have a lot of inertia.[4] Even the idioms for selection in contemporary viewers have a resolution (understandably enough) of the character level. It is impossible, for instance, to select just an accent (without also getting the letter to which it is attached). Even generating kerning tables for systems like zebrackets is somewhat daunting, suggesting the need for algorithmic kerning.

5.4 Charactles

Authors Mancilla and Plaice [MP12] proposed the *charactle* — a portmanteau word combining character and tuple — as a generalization of characters and glyphs. A charactle consists of an index into a dictionary, along with some variant or versioning information; it incorporates the Unicode character as a special case. According to this model, a text would be a sequence of charactles. The zebrackets presented in this paper are completely consistent with this approach.

[3] `http://vecg.cs.ucl.ac.uk/Projects/projects_fonts/projects_fonts.html`

[4] Of course, for specialized purposes, such as display fonts and "Word Art" (as in Microsoft Word or PowerPoint), characters are unique. These are sort of "one-off"s, with no attempt to optimize their rendering by caching them into OS tables: singletons meant to be seen as much as read, leaning towards the pictorial and away from the purely textual.

5.5 The future of literacy

The "take home message" is not only the extensibility of parentheses and brackets, but the ability to articulate any character, like a metaMETAFONT. Every glyph, stroke, mark, and pixel should be deliberately and explicitly determined for the exact circumstances of its apprehension. A character set should not be precompiled as an operating system resource, a cache of common letter forms. Such a model patronizes characters by treating them as cliches, overused forms of expression. Digital typography, electronic publishing, and computer displays allow generalization of such forms by considering characters as semi-custom instances of a richly expressive class, with factory (instantiation) specifications including not only such qualities as font family, size, and magnification, but also optical balance, reader characteristics and preferences, and arbitrary relations with any other document qualities, a kind of negotiation between aspects. Factors related to reading in the context of ubiquitous computing ("ubicomp") and IoT ("internet of things") — such as ambient illumination, whether a reader is wearing glasses or not, and time of day — should be referenced as parameters to optimize legibility and experience [Coh14]. Such exponential explosion of expression space, a hoisting of a quantized model into a seemingly continuous one, can still run on a digital computer but requires virtually arbitrary smoothness, promoting, as it were, integers into reals, necessitating on-the-fly compilation. Such display is optimally realtime, but need not be, since a document browser could initially display unadorned versions of characters, perhaps preconditioned to reflect anticipated layout, dynamically and progressively refreshing by swinging in the embellished versions as they are generated.

References

[AB89] Jacques André and Bruno Borghi. Dynamic fonts. In Jacques André and Roger D. Hersh, editors, *Raster Imaging and Digital Typography*, pages 198–204. Cambridge University Press, 1989. ISBN 0-521-37490-1.

[And90] Jacques André. The Scrabble font. *The PostScript Journal*, 3(1):53–55, 1990.

[Ann74] Mitsumasa Anno. *ABC Book* (in Japanese). Tankobon, 1974. ISBN-10 4-8340-0434-1, ISBN-13 978-4834004342.

[AO89] Jacques André and Victor Ostromoukhov. Punk: de METAFONT à PostScript. *Cahiers GUTenberg*, 4:123–28, 1989.

[CK14] Neill D. F. Campbell and Jan Kautz. Learning a manifold of fonts. *ACM Trans. Graph.*, 33(4):91:1–91:11, July 2014.

[Coh92] Michael Cohen. Blush and Zebrackets: Two Schemes for Typographical Representation of Nested Associativity. *Visible Language*, 26(3+4):436–449, Summer/Autumn 1992. `http://visiblelanguagejournal.com/issues/issue/98/`.

[Coh93] Michael Cohen. *Zebrackets*: A Pseudo-dynamic Contextually Adaptive Font. *TUGboat*, 14(2):118–122, July 1993. `http://tug.org/TUGboat/tb14-2/tb39cohen.pdf`.

[Coh94] Michael Cohen. Adaptive character generation and spatial expressiveness. *TUGboat*, 15(3):192–198, September 1994. Proceedings of the 1994 TUG Annual Meeting, Santa Barbara, CA. `http://tug.org/TUGboat/tb15-3/tb44cohen.pdf`.

[Coh14] Michael Cohen. From Killing Trees to Executing Bits: A Survey of Computer-Enabled Reading Enhancements for Evolving Literacy. In *VSMM: Proc. Int. Conf. on Virtual Systems and Multimedia*, Hong Kong, December 2014. `http://www.vsmm2014.org`.

[dK25] Guillaume Apollinaire (Wilhelm Apollinaris de Kostrowitzky). *Poémes de la paix et de la guerre 1913–1916 (Poems of war and peace 1913–1916)*. Nouvelle Revue Française, Paris, 1918, 1925.

[FVJ11] FL@33, Tomi Vollauschek, and Agathe Jacquillat. *The 3D Type Book*. Laurence King Publishing, 2011. ISBN-10 1856697134, ISBN-13 978-1856697132.

[Har07] Yannis Haralambous. *Fonts & Encodings*. O'Reilly, 2007. ISBN-10 0-596-10242-9, ISBN-13 978-0-596-10242-5.

[HF13] Steven Heller and Louise Fili. *Shadow Type: Classic Three-Dimensional Lettering*. Princeton Architectural Press, Thames and Hudson Ltd., 2013. ISBN-10 1616892048, ISBN-13 978-1616892043.

[Kar15] Peter Karow. Digital typography with Hermann Zapf. *TUGboat*, 36(2):95–99, 2015. `http://tug.org/TUGboat/tb36-2/tb113zapf-karow.pdf`.

[KH08] Robert Klanten and Hendrik Hellige, editors. *Playful Type: Ephemeral Lettering & Illustrative Fonts*. Dgv, 2008. ISBN-10 3899552202, ISBN-13 978-3899552201.

[Len91] John Lennard. *But I Digress: Parentheses in English Printed Verse*. Oxford University Press, 1991. ISBN 0-19-811247-5.

[MP12] Blanca Mancilla and John Plaice. Charactles: More than characters. In Cyril Concolato and Patrick Schmitz, editors, *ACM Symposium on Document Engineering*, pages 241–244. ACM, 2012.

◇ Michael Cohen
University of Aizu, Japan
`mcohen (at) u-aizu.ac.jp`

◇ Blanca Mancilla
Mentel, Montreal, Canada
`blancalmancilla (at) gmail.com`

◇ John Plaice
Grammatech, Ithaca, USA;
UNSW, Sydney, Australia
`johnplaice (at) gmail.com`