Omega becomes a texteme processor

Yannis Haralambous, Gabor Bella

▶ **To cite this version:**

Yannis Haralambous, Gabor Bella. Omega becomes a texteme processor. EuroTEX 2005, Pont-à-Mousson, 2005, Mar 2005, Pont-À-Mousson, France. hal-02169904

HAL Id: hal-02169904

https://hal.science/hal-02169904

Submitted on 1 Feb 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Omega Becomes a Texteme Processor

Yannis Haralambous
ENST Bretagne
yannis.haralambous@enst-bretagne.fr
http://omega.enstb.org/yannis

Gábor Bella
ENST Bretagne
gabor.bella@enst-bretagne.fr

Characters and Glyphs

The distinction between "characters" and "glyphs" is a rather new issue in computing, although the problem is as old as humanity: our species turns out to be a writing one because, amongst other things, our brain is able to *interpret* images as symbols belonging to a given writing system. Computers deal with text in a more abstract way. When we agree that, in computing, all possible "capital A" letters are represented by the number 65, then we cut short all information on how a given instance of capital letter A is drawn. Modern computing jargon describes this process as "going from glyphs to characters." If a *glyph* is the image of a writing system's atomic unit, a *character* is an interpretation of that image, an interpretation shared by many glyphs drawn by different people in different places at different times. If all these drawings are equivalent in terms of interpretation, we can consider character as an equivalence class of glyphs. To be operational such an equivalence class must be described in a clear and unambiguous way. This is why we define *character* as being a *description of an equivalence class of glyphs* [7, pp. 53–58], [6].

Arabic text provides a typical illustration ground for the concepts of character and glyph. In Arabic alphabet, letters are contextual, in the sense that a given letter will change form according to the presence or absence of other surrounding ones. When we refer to an Arabic letter and represent it graphically, we use the isolated form. We can also refer to it by its description (for example: ARABIC LETTER JEEM) and this can be considered as description of a "character": the equivalence class of shapes this letter can take in millions of Arabic documents. While there may be millions of instances of this letter, according to Arabic grammar they all belong to one of only four forms: isolated ج, initial جـ, medial ـجـ, or final ـج. Hence, we could choose to have not one but four equivalence classes of shapes: ARABIC INITIAL LETTER JEEM, ARABIC MEDIAL LETTER JEEM, and so on. But are these "characters"?

Answering to this question requires a pragmatic approach. What difference will it make if we have one or rather four characters for letter *jeem*? There will indeed be a difference in operations such as searching, indexing, etc. A good question to ask is: "when I'm searching in an Arabic document, am I looking for *specific forms* of letters?" Most of the time, the answer is negative.[1] Form-independent searching will, most of the times, produce better results and this implies that having a single character for all forms is probably a better choice.[2]

Unicode is a *character encoding*. In other words, it contains *descriptions of characters* and tries hard to define characters properly by avoiding dependence on glyphs.[3]

[1] Arabic words are not always visually segmented as English ones—there is, for example, no guarantee that the first letter of a word will always be in initial form: if a word starting with *jeem* is preceded by the definite article *al*, then the *jeem* will end up being in medial form.

[2] Greek is different: *sigma* doesn't "become" final because it "happens" to be at the end of a word. While medial *sigma* can appear anywhere, final *sigma* is used mainly for the endings of particular grammatical forms and in onomatopeias or foreign words. One would hardly ever search for *both* the final and medial form of *sigma* since their rôles are distinct. To illustrate this, when we abreviate a word by a period at a *sigma* then the latter does remain medial despite being the final letter: φιλοσοφία → φιλοσ. Hence it is quite logical to use distinct characters for medial and final *sigma*.

[3] This is not always the case because of Unicode's tenth founding principle, namely convertibility of legacy encodings—and legacy encodings contain all kinds of things. For example, again in the case of Arabic, the main Unicode Arabic table indeed contains only form-independent "characters." But, hidden towards the end of the first Unicode plane, one finds several hundreds of codepoints containing Arabic letters and ligatures in fixed forms, for legacy reasons. Like human history (or Stephen King's movies) Unicode has shadowy places which people try to avoid and even to forget that they exist.

The character vs. glyph issue is far from being solved. In this paper we give an attempt to transcend it by introducing a new concept: the *texteme*.

> A *texteme* is a set $\{c, p_1 = v_1, \ldots, p_n = v_n, g\}$ where c is a character, g a glyph, and p_i an arbitrary number of named properties taking values v_i. Character, glyph, number of properties, their names and their values can be changed at any time, by special syntax in the input file, or by OTPs, or by interaction with fonts.

The term "texteme" is sporadically used in rhetorics and other human sciences but, AFAWK, does not seem to have a well established definition. We have chosen it (it has been suggested by Patrick Andries whom we wish to thank for his help) because the ending -eme reflects the meaning of an "atomic unit of text" in the same way as in words "phoneme," "grapheme," "lexeme," etc. In the first version of this paper (included in the preprints of EuroTEX'05, we used the term "sign" instead.

In Principio Creavit Knuth TEXum

How does TEX deal with characters and/or glyphs?

First of all, `.tex` files contain characters. When TEX reads a file, it converts the data stream into tokens. A *token* ([10, §289] or [8], which is an exegesis of Knuth's B) is either a "character token" (that is, two numbers: a character code and the character's "category," which provides the specific rôle played by the given character code, for example whether it is a math mode espace character like $, or a comment escape character like %, or a group delimiter like {, and so on), or a "control sequence token."

If we leave aside for a moment the fact that TEX cannot read character codes above 255, one could claim that "character tokens" can still be considered as "characters." What happens next?

Parsing these tokens, TEX builds node lists (horizontal and vertical). A *node* is a typed atomic unit of information of a list. The amount and nature of data contained in nodes depend on their type. A "character node" [10, §134] (or "charnode") is made of two numbers: a font ID and the position of the glyph in the font table. But the latter is not bound to have any relation whatsoever with its character code. Obviously, we can hardly talk about *characters* at this point: we have crossed the bridge to Glyphland.

Another very interesting node type is the "ligature node" [10, §143]. This one contains a font ID, a glyph position in the font table, and a pointer to a linked list of charnodes. This list is in fact the "decomposition" of the ligature. TEX needs it in case it has to "break" the ligature during paragraph building, for example when a word needs to be hyphenated inside a ligature.

Talking about hyphenation, there is a node called "discretionary node" [10, §145]. This node contains two pointers to horizontal lists, as well as an integer. These horizontal lists are what is typeset when we break a word, before and after the line break (in standard cases the first one contains only a hyphen, and the second one is empty). The integer is the number of nodes of the main horizontal list to delete if the word is hyphenated (in other words: how many nodes to replace by the two horizontal lists we mentioned).

As we see, in the three node types described above only glyphs are used—never characters. There seems to be a contradiction with the very nature of hyphenation: after all, words are hyphenated according to rules given by natural language grammars, and these grammars apply to characters, not to glyphs. Indeed, would you hyphenate a word differently if some letters had calligraphic shapes? Certainly not, but for TEX, these letters are glyphs in font tables, and if variant forms exist, then their positions in the font tables are necessarily different from the standard ones. How does TEX deal with this?

There is in TEX a primitive called `\lccode` (and a corresponding `WEB` macro called *lc_code*). Each glyph in a font participating in hyphenation has necessarily a *lc_code* value. These values are usually initialized in the format.

lc_code is in fact a mapping between glyphs and characters. Hyphenation rules are written using patterns, and patterns use characters. When TEX needs to hyphenate words in a paragraph, it first maps glyphs back to characters using *lc_code* [10, §892–899], and then applies hyphenation rules.

This method seems to work, but the user must, at all times, use the appropriate *lc_code* settings for each font.[4]

Let us continue our journey through TEX and see what happens in the final stage. There is no surprise: the information contained in charnodes (namely font ID and glyph position) is output to the DVI file [10, §619]. Ligature nodes are treated

[4] It is worth mentioning that *lc_code* has a big advantage after all: it allows *simplification* of hyphenation patterns. Indeed, instead of mapping a glyph to the precise character it represents, one can use equivalence classes of characters. For example, in Greek, hyphenation does not (or very rarely) depend on accents and breathings, so we can map all letters with diacritics into base letter classes and write patterns using the latter.

similarly [10, §652]: only font ID and glyph position of the ligature remains, provided of course that the ligature has survived hyphenation. Discretionary nodes vanish long before output since either hyphenation has occured and the two horizontal lists pointed by the node have found their way into the DVI file, or no hyphenation has occured and the discretionary node falls into oblivion.

When TeX and the DVI file format were developed, this was the best possible approach: DVI files had to be short, and there was no need to insert more information than required to print. And indeed, a printing device doesn't care whether a glyph is a ligature or whether it derives from hyphenation; printing is done in Glyphland, at the very end of the line of document production process. Even PostScript language didn't change that situation, although it made printing devices more clever (clever enough to interpret a very rich prgramming language).

Dixitque Berners Lee: fiat Web et facta est Web

Things changed when DVI and PostScript were not anymore the only targets of TeX document production process. The Web brought the era of electronic documents in various formats such as PDF, XHTML, etc. These documents allow interaction with textual contents: copy-and-paste of text blocks, searching, indexing, etc.

When we search for a word inside a document, or in a collection of documents, do we care about the shape of its letters? Most of the time, the answer is no. Otherwise, it would be quite hard to find a word in a document written in *Zapfino* or *Poetica*, since one has to predict the precise variant form used for every letter, and there are many of them.

In this kind of situation one would like to interact with the document on the character level. But if PDF or XHTML files are produced by TeX, then the information on characters is lost. A very simple example: if 'fi' is represented in a DVI file as glyph 12 of font cmr10, with no reference whatsoever to the character string 'f-i', then how on earth can we search for the word 'film' by entering characters 'f', 'i', 'l', 'm' in our program's search interface?

There is no natural solution to this problem. Acrobat Distiller tries to give an algorithmic solution by using PostScript font glyph names ([7, pp. 651–653], [1]). The idea is the following: in PostScript type 1 fonts, glyphs have names (namely the names of PostScript subroutines which contain the Type 1 operator glyph's description); when creating a variant glyph of, let us say, letter 'e', designers are requested to use a name like `e.foo` where `foo` is some description of the variant: the first part of the name identifies the Unicode character and the second, the variant; Distiller goes through all glyph names in all fonts used in a document and maps glyphs to Unicode characters according to their names. There is a similar syntax provided for ligatures (that is: glyphs mapped to more than one Unicode character).

TrueType fonts have a table (called `cmap` [7, pp. 703–706]) dedicated to this mapping: we map (single) Unicode characters to (single) glyphs.[5]

These solutions are sub-optimal. There is no way to modify the mapping between characters and glyphs without hampering with the font, and this is not always desirable.

Instead of finding sub-optimal solutions to a problem which is the consequence of information loss in the DVI file, let us attack the origin of this problem. Is it possible to keep character *and* glyph information all the way long, from input file to DVI (and beyond)?

"How now, spirit! whither wander you?" (Enters Omega$_1$)

One of Omega$_1$'s goals was to achieve Unicode compliance. The least one could expect of Omega$_1$ is an affirmative answer to the final question of previous section: Can we obtain Unicode information in a DVI file?

Before answering that question let us see whether Omega$_1$ is actually different from TeX when dealing with characters and glyphs. It isn't: Omega$_1$ can read 16-bit characters (some versions of it can even read UTF-8 representation of Unicode data directly), but once inside Omega$_1$, Unicode characters become "character tokens" and then charnodes, ligature nodes and discretionary nodes all the same as in TeX.

How then does Omega$_1$ manage to do Arabic, a writing system where one just *has* to go from characters to glyphs? By using OTPs [9]. An OTP is an internal filter, applied to "character tokens." It can be compared to a pre-processor but has some major advantages: the fact that only tokens are targeted (not comments, for example), and that the catcode of each token is known and that transformations

[5] In fact, things are worse than for PostScript Type 1 fonts: while PostScript glyphs have names (and names are usually meaningful and stable vs. font trasformations), TrueType glyphs are accessed by their "glyph index values" which are plain integers. A TrueType font opened and saved by some font editing software can be re-organized, glyph index values can change without further notice, and hence accessing a glyph directly by its index, without going through the `cmap` table, is quite risky.

are applied to selected categories only (usually plain text, that is: catcodes 11 and 12). Furthermore, OTPs have the tremendous advantage of being dynamically activated and de-activated by primitives.

Let us analyse the example of Arabic typesetting via Omega$_1$. When Arabic Unicode characters are read, they become "character tokens" (the first part of the token takes the numeric value of the Unicode codepoint, the second part is the catcode, in this case probably 12). No contextual analysis is done yet. It is an OTP that analyses the context of each glyph, and, using a finite-state machine, calculates its form; the result of the transformation by this OTP is one or more new tokens, replacing the previous ones. These tokens correspond to given forms of glyphs. Other OTPs will act upon them and produce esthetic ligatures, and usually the final OTP will map these tokens to font-specific ones, which in turn will become charnodes, and will end up in the DVI file.

The purpose of keeping only the last OTP font-dependent is to improve generality and re-usability of the previous OTPs. But from the moment we perform contextual analysis we have left Unicode data behind and are travelling in a no-man's land between characters and glyphs. In this de Chirico-like surreal place, characters are more-or-less "concrete" and glyphs more-or-less "abstract." Obviously, if the result is satisfying—and this is the case with Omega$_1$'s Arabic typesetting—it is of no importance how we manage to obtain it, whether we go through these or those OTPs and in which ways we transform data.

But the fact is that we do lose character information, just as in TeX. In the DVI file we have beautiful Arabic letters and ligatures ... but there is no way back to the original Unicode characters.

This situation is changing with Omega$_2$ (work in progress). Instead of characters, character tokens and charnodes we are using textemes (texteme tokens and texteme nodes), links and bifurcations. Texteme nodes are data structures containing a character, a glyph, and additional key/value pairs, where the value can be simple or complex, involving pointers to other textemes, etc. Links are groups of textemes which contain alternative sets of glyphs based on a graph: the paragraph builder includes this graph into its own acyclic graph through which it will obtain the optimal paragraph layout as the shortest path from top to bottom.

Before we enter into the details of textemes, let us briefly describe another paradigm of character/glyph model implementation: SVG.

The SVG Paradigm

SVG (= Scalable Vector Graphics, [4]) has at leats one very nice property: the way text is managed is quite elegant.

First of all, an SVG document is an XML document, and the only textual data it contains is actual text displayed in the image. In other words: however complex a graphic may be, all graphical elements are described solely by element tags, attributes and, eventually, CDATA blocks. Not a single keyword will ever appear in textual content.

This is not at all the case of LaTeX, where we are constantly mixing mark up and contents, as in:

```
Dieser Text ist \textcolor{red}{rot}.
```

where `red` is markup and `rot` is text, and where there is no way of syntactically distinguishing between the two. In SVG, this example would be:

```
<svg:text>
Dieser Text ist
<svg:tspan color="red">rot</svg:tspan>.
</svg:text>
```

where separation between text and markup is clear.

In SVG, as this is the default for XML, text is encoded in Unicode. In other words, text is made of characters only. How then do we obtain glyphs?

As in TeX, SVG uses the notion of "current font," attached to each `text` or `tspan` element, and this informations is inherited by all descendant elements, unless otherwise specified. Fonts can be external, but the model is even more elegant when fonts are internal.

An internal SVG font is an element called `font` containing elements called `glyph`. The latter has an attribute called `unicode`. This attribute contains the one (or more, in case of a ligature) Unicode characters represented by the glyph.

The contents of the glyph element can be arbitrary SVG code (this is quite similar to the concept of TeX's virtual fonts, where a glyph can contain an arbitrary block of DVI instructions). In this way, any SVG image, however complicated, can become a single glyph of a font. To include this glyph in the SVG graphic one only needs to select the font and ask for the same Unicode character sequence as in the `unicode` attribute of the glyph, in a `text` element.

Besides `unicode`, the `glyph` element takes a number of attributes :

- `glyph-name`: in case we want to access a glyph directly (useful when we have more than one variant glyphs representing the same character);

- `d`: the actual outline of the glyph, if we want to keep it simple and not include arbitrary SVG graphical constructions as contents of `glyph`;
- `orientation`: when mixing horizontal and vertical scripts, how is this glyph oriented?
- `arabic-form`: initial, medial, isolated or final?
- `lang`: use this glyph for a number of given language codes only;
- `horiz-adv-x`, `horiz-adv-y`: the advance vector of the glyph when typeset horizontally;
- `vert-adv-x`, `vert-adv-y`: idem, when typeset vertically;
- `vert-origin-x`, `vert-origin-y`: the origin of the glyph when typeset vertically.

What happens if we want a specific glyph, other than the standard one obtained directly going through Unicode character sequences? We can use element `altGlyph` which allows "manual" insertion of glyphs into text, and the PCDATA contents of which is the Unicode character sequence corresponding to the alternate glyph (in this way we get, once again, textual data for indexing, searching, etc.). But `altGlyph` also takes some attributes:

- `xlink:href`: if the font is described as an SVG `font` element, an XLink reference to the corresponding glyph element inside the font—our way of going directly to the glyph, no matter where it is located: server on the Web, file, font;
- `format`: the file format of the font (SVG, OpenType, etc.);
- `glyphRef`: a reference to the glyph if the font format is other than SVG (the specification provides no additional information, we can reasonably assume that this could be the PostScript glyph name in case of CFF OpenType fonts, or the glyph index value in case of TrueType-like fonts, but, as we said already, this is quite risky);
- `x` and `y`: if the alternate glyph is indeed typeset, then these should be the absolute coordinates of its origin;
- all usual SVG attributes (style, color, opacity, conditionality, etc.).

Let us suppose, for example, that we want to write the word "Omega" with a calligraphic 'e' (font *Jolino*) described in the element:

```
<svg:glyph unicode="e" glyph-name="e.joli"
   d="... its path ..."/>
```

We only need to write:

```
<text>
   <tspan font-family="Jolino">
   Om<altGlyph
   xlink:href="#e.joli">e</altGlyph>ga
   </tspan>
</text>
```

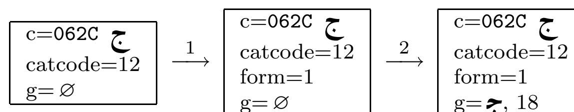
We can conclude by saying that SVG jolly well separates textual contents from markup (characters are provided as contents, glyphs as markup), and that `altGlyph` element comes quite close to the goal of our notion of texteme: it provides both a character (in fact, one or more characters), a glyph, and some additional properties expressed by attributes. These are not really entirely user-definable as in the case of texteme properties, but one could very well introduce additional attributes by using other namespaces.

Introducimus Textema

In the remainder of this paper we will describe the texteme concept in more detail and give some examples of applications. We will use the notation [c=0061 a / g= a, 97] for a texteme containing character U+0061 LATIN LETTER A, glyph "a" (position 97 in the current font), and no additional properties.

Using this notation, an initial Arabic *jeem* would need a texteme [c=062C ج / form=1 / g= ﺟ, 18]. If we would like to typeset this texteme in red color, we would add another property: [c=062C ج / form=1 / color=red / g= ﺟ, 18].

Here is how it happens: Omega$_2$ reads a file containing Unicode character U+062C. Tokenisation produces texteme [c=062C ج / catcode=12 / g= ∅] (no glyph for the moment). Then we go through the OTPs for contextual analysis and font re-encoding:



The first OTP provides the contextual form value, without affecting the character (or catcode) value. The second OTP adds the glyph information (which would otherwise be added implicitly when reading the font data).

Now what if we wanted the glyph to be typeset in red? All depends if we want (a) only the specific instance of texteme to be red, or (b) all *jeem* characters, or (c) all initial *jeem* characters. In the first case one would manually change the `color` property

of this texteme to value `red`.[6] In the second case one would use an OTP matching all textemes conforming to the pattern [c=062C ج / * / g= *] (where asterisks indicate arbitrary values), and would add (or modify the value of) property `color`. In the third case one would, again, use an OTP but this time matching textemes conforming to the pattern [c=062C ج / form=1 / * / g= *]. That way only initial form textemes will be catched.

One can also imagine OTP patterns based solely on properties. If we wanted to change the color of all red glyphs into green, we would use an OTP as the following:

c=* color=red * g= *	→	c=(same) color=green (same) g= (same)

Besides catcode, (Arabic) form and color, one can imagine many other properties: horizontal and vertical offset, hyphenation (or hyphenation preference), penalty, glue, bidirectionality level, language, style, word boundary, etc. We will discuss them while describing selected examples of texteme applications.

Locked Properties Whenever we define rules, we also need ways to allow exceptions. In our previous color example, let us suppose that there is a given texteme which has to remain blue, despite all OTPs which will try to change its color. Properties can be locked: if [c=ج / form=1 / color=blue / g=ج, 18] becomes [c=ج / form=1 / 🔒 color=blue / g=ج, 18], then no OTP will ever be able to change this property. Of course, OTPs can lock 🔒 and unlock 🔓 properties, so if that color has to be changed after all, then it can always be unlocked, modified, and locked again . . .

Textemes in Auxiliary Files What's the use of having textemes and texteme properties if all the information is lost when tokens are written into a file? For example, when textemes happen to be in the argument of a `\section` command which will be written in a `.toc` file. Instead of losing that information we will write it into that file (which becomes a *texteme file*), and have Omega$_2$ read it at the subsequent run and import textemes directly.

Texteme Documents And since Omega$_2$ reads and writes auxiliary texteme files, why not input the main file as a texteme document? One could imagine a texteme-compliant text editor, a kind of super-Emacs in which one would attach texteme information (characters, glyphs, predefined or arbitrary properties) to the TEX code. One can imagine how simple operations like the verbatim environment would become: if we can fix the catcode of an entire text block to 12, then all special characters (braces, backslashes, percents, ampersands, hashes, hats, underlines) lose their semantics and become ordinary text, LATEX only needs to switch to a nice typewriter font and use an `\obeylines` like command and we're done.

Such a text editor is nowadays necessary when we are dealing with OpenType fonts requiring interaction with the user. For example, the `aalt` feature [7, p. 798] allows choosing variant glyphs for a given character. It would be much more user-friendly to use a pop-up menu than writing its glyph index value in the TEX code. That pop-up menu would insert the glyph index as a texteme property, and bingo.

Explicit Ligatures

To understand how we deal with ligatures let us recall how TEX uses them in the first place. When building the horizontal list (this part of code is called the chief executive) every charnode is tested for the presence of eventual ligatures (or kerning pairs) in the font's lig/kern program [10, §1035]. If a ligature is detected then a ligature node is created. There is a special mechanism to ensure that the created ligatures is always the longest one (so that we get 'ffl' instead of an 'ff' followed by an 'l').

This ligature node contains a pointer to a horizontal list containing the original charnodes. These will be used in the paragraph builder if hyphenation is necessary.

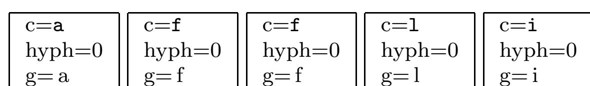
If we need to hyphenate inside a ligature then it the lignode is first disassembled into the original charnodes [10, §898] and then a discretionary node is created with the two parts of the ligature as pre-break and post-break lists [10, §914]. This approach allows only one possible hyphenation inside a

[6] Why should we insert the color information into the texteme as a property, when we can simply use a macro like `\textcolor`? Because OTPs use buffers and control sequence tokens and character tokens of categories other than 11 and 12 will end the buffer and send the buffered text for processing. If the buffer happens to end inside an Arabic word, then there is no way to do proper contextual analysis since the OTP cannot know what will follow in the next buffer. The only way to obtain a texteme string sufficiently long to perform efficient contextual analysis, is to avoid control sequence tokens inside Arabic words, and this is easily achieved by storing information in properties.

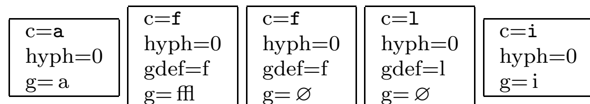
ligature—as Don says in [10, §904]: "A further complication arises if additional hyphens appear [...] TeX avoids this by simply ignoring the additional hyphens in such weird cases." This can be quite annoying for ligatures of 4 or more letters containing 2 or even 3 potential hyphenation points, and due to the increasing popularity of OpenType fonts we will get more and more of such ligatures in the future.

In TeX a ligature can be inserted *only* by reading the font lig/kern program. It is impossible to instruct TeX to create a ligature node using control sequences or other means, and hence it is equally impossible to do it in Omega_1 using OTPs.

Our goal is to do this in Omega_2, using textemes. We call such a ligature an *explicit* one (in contrast to *implicit* ligatures contained in fonts). Let us take the example of the 'ffl' ligature in the word "affligé." Let us first suppose that there is no hyphenation point in this word:



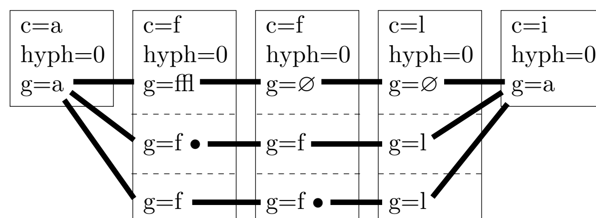
To insert a ligature one would replace it by:



In this string, character information is left untouched and the ligature glyph is placed in the first texteme participating to the ligature (the remaining ones have void glyphs). The **gdef** properties contain the "default glyphs," in case the ligature is broken.

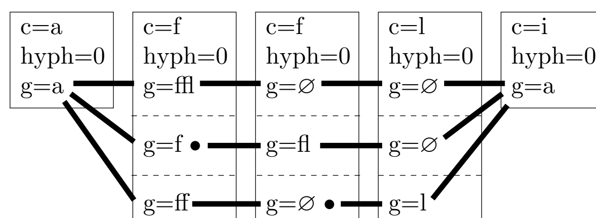
This brings us to a new notion, the one of *link*. The texteme string shown in the previous example is, in fact, a doubly linked list. A link is a set of doubly linked textemes, in our case those producing the ligature. We say that they participate to the link. The reason for linking these textemes is that, at any moment, some OTP may insert additional textemes between the ones of the link. We have to be sure that when these textemes arrive to the paragraph builder, they will produce a ligature only if they are still consecutive, otherwise we will fall back to the default glyphs.

Things get more complicated if there is a hyphenation point. In this case we must provide all possible combinations of ligatured and non-ligatured glyphs. These combinations form an acyclic graph, very much like the one of TeX's paragraph builder, we call it a set of *bifurcations*. In the figure below, we have illustrated a quite complex case: a ligature 'ffi' surrounded by letters 'a' and 'i' and containing two hyphenation points (after the first and the second 'f' letter), a mission impossible for TeX:



The fat strokes in the figure are the vertices of the graph. These vertices will be examined later for eventual kerning pairs or for other ligatures. The bullet after a glyph indicates that at this location we have a mandatory line break.[7] Notice that all **hyph** properties are now set to 0 since the discretionary hyphenation is handled "manually" by bifurcation.

Here is the same figure, completed with 'ff' and 'fl' ligatures which will only be used in cases the original 'ffl' is broken:



Let us not forget that this graph deals only with glyphs. Characters still form a plain (one-dimensional) string, and macro expansion will use textemes in exactly the same way as it currently uses character tokens. The paragraph builder, on the other hand, will include this graph as a subgraph of its network for finding the shortest path. Where we have placed a bullet, the paragraph builder will consider it as a mandatory end-of-line preceded by a hyphen glyph.

Non-Standard Hyphenation

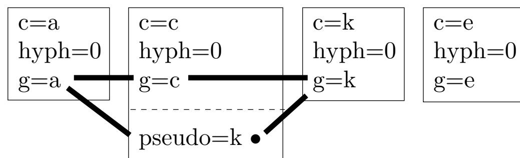
Standard hyphenation corresponds to TeX's `\-`: the first part of the word stays on the upper line and is followed by a hyphen glyph but otherwise unchanged, and the remaining part is placed on the lower line, also unchanged.

But "there are more things in heaven and earth, Horatio." Typical examples of deviant hyphenation are German words containing the string 'ck' (which, together with 'ch' is a ligature in traditional German typography in the sense that glyphs are brought closer to each other) or Hungarian 'ssz' which in

[7] The purpose of this bullet is to postpone until the very last moment the creation of a texteme [c=∅ g=-] followed by a line break. The user should be able to decide whether character properties of line break hyphens should be void or U+00AD SOFT HYPHEN, or ant other character.

some cases is hyphenated 'sz-sz' (*ösz-sze*) and in other cases (when the word is composite, like in *kis-szerű*) 's-sz'.

Obtaining this using bifurcation is very easy:



The paragraph builder will have to choose between an unbroken glyph string 'ack' and a string 'ak' followed by a hyphen, a line break, and another 'k.' We can insert this information in textemes very early, it will remain alive until the paragraph builder. On the character level we keep 'ack' so that in text extraction or in conversion to a file format without explicit line breaks (like XHTML) we will always keep the regular 'ack', whether or not there has been hyphenation in the DVI file.

There are similar phenomena involving punctuation or diacritics: in Polish, when a word is broken after an explicit hyphen, then we get a hyphen at the end of line, and another hyphen at line begin. In Dutch, 'oe' is pronounced 'ou' unless there is a diaeresis on the 'e'; when a word is broken between 'o' and 'ë', then the diaeresis disappears (since breaking the word at that point makes it clear that the two letters do not form a diphthong). In Greek we have exactly the same phenomenon as in Dutch.

It should be interesting to note that this situation of discrepancy between visual information and text contents is being taken into account by formats like PDF. Indeed, version 1.4 of PDF has introduced the notion of *replacement text* where one can link a character string (← the characters) with any part of the page contents (← the glyphs) [2, p. 872]. The example given is the one of German 'ck' hyphenation:

```
(Dru) Tj
/Span
<</ActualText (c) >>
BDC
(k-) Tj
EMC
(ker) '
```

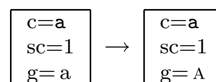
The `ActualText` operator specifies that the string "c" is a "logical replacement" for the contents of the `BDC`/`EMC` block, which contains precisely the string "k-." As we see, using texteme properties to keep this particular information until the DVI file (and beyond) makes sense since PDF is already prepared for handling it, and by using it one can enhance user-interaction with the document.

OpenType Features

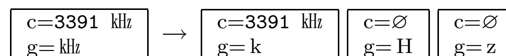
OpenType fonts contain information on various glyph transformations. This works roughly in the following way: the user activates "features," for each feature the font attempts "lookups" (pattern matching on the glyph string), and for each matched lookup there is a series of glyph positionings or glyph substitutions. In our case, activated features become texteme properties (so that they can be entered and modified at any time, independently of macro expansion, and so that they are carried over when tokens are stored, moved or written to files), then at some point, chose by the user, lookups are applied to texteme strings, and the effect of positionings and substitutions is again translated into texteme properties, before the textemes arrive to the paragraph builder.

Both glyph substitution and positioning act on the glyph part of textemes only. Let us review briefly the different types of OpenType transformations [7, p. 746–785]:

- *single substitution*: a glyph is replaced by another glyph. For example, a lowercase letter is replaced by a small cap one;

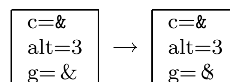


- *multiple substitution*: a glyph is replaced by more than one glyphs. For example, we may want to replace the ideographic square ㎑ by the glyph string "kHz":

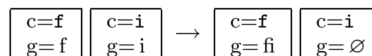


We generate additional textemes with empty character parts so that eventual interaction between the glyphs of these textemes is possible (for example, they may kern or be involved in some other OpenType transformation).

- *alternate substitution*: one chooses among a certain number of alternate glyphs for a given texteme. The user provides the ordinal of the desired variant glyph as a texteme property:



- *ligature substitution*: the ordinary ligature. Once again we have to use glyph-less textemes:



- *contextual substitution*, *chaining contextual substitution*, *reverse chaining contextual substitution*: meta-substitutions where one has a

pattern including glyphs before and/or after the ones we are dealing with, with eventual backtrack and lookahead glyph sequences, and sometimes going from end to start;

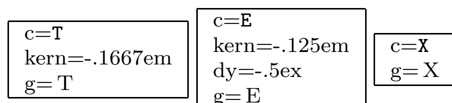
- *single adjustment*: this is a transformation that adjusts the position of a glyph. In TeX, you can move a glyph horizontally by using `\kern` of `\hskip` commands, and vertically by putting it in a box and `\raise`-ing or `\lower`ing that box. In both cases you lose hyphenation and kerning, and since control sequence tokens are involved, OTP buffers are terminated.

  This is why we introduce two very important texteme properties: `dx` and `dy`. They provide horizontal and vertical offsets without going through control sequence tokens. There is no boxing, the advance vector of the glyph does not change, and the effect of moving the glyphs around does not affect surrounding boxes. In other words: even if you raise a glyph using `dy`, this will not affect your baseline—it is rather like if you had used a `MOVEUP` instruction in a virtual font.

  Our favourite example of such a transformation: the TeX logo (one of the first things people learn about TeX, since it is described on page 1 of the TeXbook) becomes a plain texteme string without any control sequence inbetween. Here is the standard TeX code, taken from `plain.tex`:

```
\def\TeX{T\kern-.1667em\lower.5ex%
\hbox{E}\kern-.125emX}
```

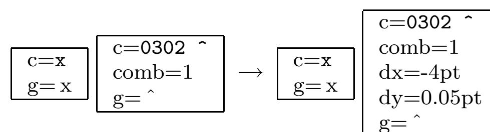
  and here is the texteme string:



  (see below for the `kern` property);

- *pair adjustment* is like single adjustment, but is applied to a pattern of two glyphs. Kerning is the most common case of pair adjustment. Besides `dx` and `dy` we also provide `kern` and `vkern` properties for this. The difference with `dx` and `dy` is that the advance vector of the glyph is modified. To see the difference, here is the TeX logo first with a `kern` property and then with a `dx` property on texteme 'E': TeX, TeX;
- *cursive attachment* is a very interesting transformation: we define a mark (that is a point in the glyph's coordinate space) on each side of a glyph, and we declare that proper typesetting in this font is achieved when the right mark of glyph n is identified with left mark of glyph $n+1$. This eliminates the need of kerning pairs (both horizontally and vertically) and is ideal for cursive fonts with connected letters (as we used to write on the blackboard in primary school). We define a property called `cursive`, when it is activated Omega$_2$ will first check that the marks exist in the font, then do the necessary calculations, and finally insert `kern` and `vkern` values to match marks;
- *mark to base attachment*: the same principle as cursive attachment, but this time the goal is to connect a "base" to an "attachment." Usually this will be the glyphs of a Unicode base character and the one of a combining character. The simplest example: a letter and an accent. TeX veterans still remember the headaches caused to Europeans by the `\accent` primitive. Since 1990, thanks to the Cork encoding and its followers, we have been able to avoid using this primitive for many languages. But there are still areas where one cannot predict all possible letter + accent combinations and draw the necessary glyphs. To make things worse, Unicode compliance requires the ability to combine any base letter with any accent, and even *any number* of accents!

  To achieve this, one once again defines marks on strategical positions around the letter (accent scan be placed above, beyond, in the center, etc., these positions correspond to Unicode combining classes) and around the accent. When the glyph of a combining character follows the one of a base character, all we need to do is find the appropriate pair of marks and identify them, using `dx` and `dy` properties. Here is an example:



  and the result is 'x̂' (we have deliberately chosen a letter-accent combination which is used in no language we know of, so that there is no chance to find an already detextemeed composite glyph in any font);

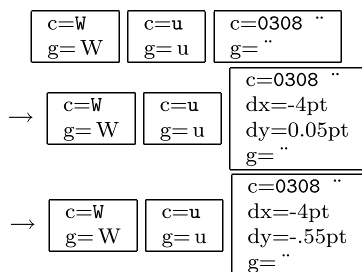
- *mark to mark attachment*: instead of attaching the glyph of a combining character to the one of a base character, we attach it to the one of another combining character. The method is strictly the same;
- *mark to ligature attachment*: same principle but things get more complicated since a ligature can have more than one marks in the same

combining class and corresponding to individual letters. The idea is to read a ligature of n glyphs followed by n combining glyphs and to place the latter on appropriate locations above (or more generally, around) the former. This is rarely encountered in Latin fonts, but becomes crucial in Arabic fonts with many ligatures (since short vowels and other diacritics are considered as combining characters by Unicode);

- *contextual* and *chaining contextual positioning*: again a meta-transformation where a pattern of glyphs is matched (with eventually a backtrack and a lookahead) and then one or more of the previous positioning transformations are applied. This is crucially missing from TeX.

  A typical example is the acronym S.A.V. (= "Service Après-Vente"), where the 'V' should be brought closer to the period preceding it because the latter is itself preceded by an 'A'. In the case of, for example, S.V.V. (= "Schweizerische Vereinigung für Vegetarismus") kerning between period and second 'V' should better not be applied.

  Another example is German word "Würze," where, in some fonts with a very expansive 'W', the Umlaut has to be brought closer to the letter to avoid overlapping 'W'. In this case we (a) match the pattern of three textemes 'Wu¨', (b) place the accent on the 'u', and (c) lower it:



Application of transformations contained in GPOS and GSUB tables will be considered like activating an OTP, so that the user may insert additional OTPs between those two, or after them.

Doing Better Than OpenType

Positioning of diacritics on Arabic ligatures [3], or of Masoretic diacritics in Biblical Hebrew [5] is a non-trivial task. There are algorithms calculating positions of diacritics using methods such as force-fields, blank area calculation, etc. Until now it is impossible to apply such algorithms without implementing them into the very kernel of TeX.

Using texteme OTPs one would first apply contextual analysis, then GSUB transformations (and GPOS for whatever it is worth) and finally, after the string chain has gone through all OpenType transformations, apply positioning algorithms as external OTPs. At that stage we know exactly which ligatures are used and what the final shape of each word is. The algorithm would obtain the glyphs of ligatures and vowels used—as well as special information such as the presence of keshideh—from texteme properties. Having access to the glyph contours of the specific font, it would then reconstruct in memory an envelope of the global graphical image of the word, containing visual centers of individual letters and other relevant information. The result of calculations would be included in dx and dy properties of vowel textemes. After that, processing would continue normally.

In fact, our approach not only uses all resources that OpenType fonts can provide but, contrarily to other systems which rely on OpenType for the final typesetting steps, it allows OTP transformations before GSUB, between GSUB and GPOS and even after GPOS. And if we want to use OpenType transformations only partially, we can always lock properties and hence avoid them to be modified by the font.

Meta-information

For TeX, the only way to include metadata (that is: information which is not part of the page description) in a DVI file is through the \special primitive. "Specials" are not supposed to interfere with typesetting, but they actually do: if we write

```
A\special{blabla}V
```

there will be no kerning between these two letters. Which means that if we want to change the color of letter 'V' only, we will lose kerning. In Omega$_1$, there is a primitive allowing us to avoid this problem: \ghost (which would emulate the behaviour of a glyph related to kerning, without actually typesetting the glyph), but this solution is rather clumsy.

Using textemes, we can insert the color information as a property and then include the necessary PostScript code for changing color, long after kerning has been applied (kerning, which, by the way, is also a texteme property), or even leave the color property in the DVI file and let (texteme-compatible) dvips read that information and write the appropriate PS code.

One could even define texteme properties having no effect whatsoever on typesetting. For example, in Arabic, one could mark the letters *alef* and *lam* of the definite article, or the letter *waw* of the "and" particle, as playing these specific grammatical rôles, so that we can easily distinguish them from letters

alef+lam or *waw* which just happen to be at the beginning of a word. The interest of this lies in the fact that Arabic is not visually separating them from the following word.

Or, again in Arabic, one could imagine a morphological analyser (acting as an external OTP) which would give the letters of the Semitic root of each word a specific texteme property. Such letters would be *al*__k__*i*__t__*ā*__b__*u* (the book), __k__*u*__t__*u*__b__*un* (books) *a*__kt__*u*__b__*u* (I write), etc. This is the kind of information which would enormously facilitate searching and indexing, but which we would like to avoid representing visually since it would only obstruct reading.

Characters, Texteme Properties or Higher Order Markup?

In the previous section we have suggested uses of texteme properties which do not affect typesetting. Most often these can also be achieved by characters or by higher order markup.

For example, Ω besides being a popular software project is also a letter of the Greek alphabet *and* the symbol for the SI unit for resistance, named after its inventor Georg Simon Ohm (1789–1854). To distinguish between these two uses of the same symbol, Unicode provides two different characters (`U+03A9` and `U+2126`). Clearly it would be preferable to use one of them should to distinguish between "Omega" and "Ohm," rather than texteme properties or higher order markup.

We mentioned the possible use of texteme properties for marking the current language. This can seem practical but also has drawbacks: languages are nested, even if their nesting is not always compatible with the logical structure of the document. It would be better to use LaTeX commands for marking languages since these commands will not interfere with micro-typography. Indeed, the author can hardly imagine the need of changing the language of a word in the very middle of it, so that we incur the danger of losing kerning or hyphenation[8]). Hence, such properties can, at first sight, very well be handled by usual higher level markup.

The best possible use of texteme properties is for cases where control sequence tokens would otherwise interfere with the very fragile operations of microtypography and hyphenation.

Glue, Penalty, CJK Languages

Be it fixed or flexible glue, it is now possible, through texteme properties, to add it to glyphs, without affecting already existing kerning (which would be added to this glue), ligatures, hyphenation, OTPs that may match the word, etc.

The typical example is letterspacing: how do you increase space between letters[9] while keeping hyphenation of the word, f-ligatures, etc.? Before Omega$_2$, to achieve this, the author was bound to define special font metrics (with tens of thousands of kerning pairs). Now it suffices to add a simple `kern` property to each texteme.

Glue for all glyphs is also required in CJK languages where there are no blank spaces between ideographs but where one sometimes needs to shrink or stretch the contents of a line because of a punctuation mark or a closing delimiter which are not allowed to be on line begin, or an opening delimiter which is not allowed on line end. So, even if this is not obvious when reading such text, we do put some glue (with a very small amount of flexibility) between *every* pair of ideographs. In Omega$_1$ this is handled by OTPs, but once such an OTP is used, the ones following it cannot match patterns of ideographs anymore because of the control sequence tokens between them. Once more, it is more natural to systematically add a small amount of glue to each ideograph, using a texteme property.

Adding glue to every ideograph is a good thing, but how do we avoid lines starting with punctuation or closing delimiters?

If we can add glue to textemes, why not penalties? In that way the space between an ideograph and a punctuation mark or a delimiter will be exactly the same as for all other ideographs, but using an infinite penalty value, line breaking will be prohibited at that point.[10] Here is an example of some ideographs and the corresponding values of glue and penalty (as used by the author):

---

[8] Although, nowadays, people use more and more language mixtures, like the notorious French *antislash* for "backslash" . . . In fact, in French one has anglicisms (French words used with their English meaning, like *librairie* for [code] library, etc.), English words that found their way into French vocabulary (*week-end*, *starlet*, etc.), English words that have been artificially gallisized (*débogage* ← "debugging," *shunter* ← "to shunt", etc.) and many other levels of interaction between the two languages. Hyphenation of these words depends on their level of French-ness, which can vary temporally and geographically.

[9] *Cave canem!* Letterspaced typesetting should be attached to specific semantics and should never be done for justification reasons only, otherwise it is like stealing sheeps.

[10] We don't have that problem in Latin typography because line breaking is allowed only at glue nodes (that is, mostly between words) and inside words using hyphenation—but a punctuation mark has no *lc_code* and hence cannot be matched by a hyphenation pattern.

c=9019 這 glue=0pt stretch=.025em g= 這	c=672C 本 glue=0pt stretch=.025em penalty=10000 g= 本	c=3002 。 glue=0pt stretch=.025em g= 。

to obtain: 這本。

Glue vs. the "Space Character"

It is well known to us TEX users, that TEX (and thus also DVI) has its own philosophy about how words are separated, namely by glue. The DVI page is like a sea of glue in which glyphs navigate and give the impression of forming words by getting closer to each other. But this is only illusion. In DVI there is no way of distinguishing between, for example, inter-word space and kerning. It is the human eye that deciphers spaces between some letters as being word separators (and the narrower these spaces are, the more difficult is reading). In other markup or typesetting systems, things are different. Unicode defines character U+0020 SPACE as well as a dozen other "whitespace characters." Some of them are extensible and others of fixed width. PostScript uses a mixed approach: when the glyph of the space character has the right width, it is used in strings; when a different width is needed, due to justification, PostScript uses horizontal skips, similar to DVI ones. But in PDF space characters must be present, since people may copy-paste text into other applications: they would be quite surprised to find blank spaces are missing ...

To be able to distinguish glue produced by interword space from glue entered explicitly, we use a texteme for interword glue. This texteme has a character part which is one of the Unicode whitespace characters and a blank glyph part. "Blank" is not the same as "void": this texteme has indeed a glyph, which can therefore be matched by OpenType lookups, but this glyph has no contour and its advance vector can vary.

Using this approach, not only can OpenType lookups match whitespace glyphs but we can also produce adequate PDF, SVG and XHTML code (for example: in XHTML, interletter kerning should be ignored but interword spaces must be kept in form of Unicode whitespace characters).

Conclusion and Caveats

Work described in this paper is experimental. In other words: what we present here is the latest status of our investigations and experimentations, in the frame of the research project INEDIT of ENST Bretagne. Our goal is to provide a new microtypographical model for typesetting (different from the node-model of TEX) which will be Unicode- and OpenType-compliant, which will provide more control to the user than any Unicode or OpenType-compliant application, and which will produce documents with sufficient information to be converted into any present or future electronic document file format.

There is a discussion list `omega@tug.org` hosted by TUG and dedicated to this project. To subscribe, please visit:

`http://tug.org/mailman/listinfo/omega`

References

[1] Adobe Systems. Unicode and glyph names, 2003.

[2] Adobe Systems. *PDF Reference: Version 1.6.* Addison-Wesley, 5th edition, 2004.

[3] Gábor Bella. An automatic mark positioning system for Arabic and Hebrew scripts. Master's thesis, ENST Bretagne, Octobre 2003.

[4] Jon Ferraiolo, Jun Fujisawa, and Dean Jackson (eds.). *Scalable Vector Graphics (SVG) 1.1 Specification.* W3C, 2003.

[5] Yannis Haralambous. *Tiqwah*, a typesetting system for biblical Hebrew, based on TEX. In *Actes du Quatrième Colloque International* Bible et Informatique*, Amsterdam, 1994*, pages 445–470, 1994.

[6] Yannis Haralambous. Unicode et typographie : un amour impossible. *Document Numérique*, 6(3-4):105–137, 2002.

[7] Yannis Haralambous. *Fontes & codages.* O'Reilly France, 2004.

[8] Yannis Haralambous. Voyage au centre de TEX : composition, paragraphage, césure. *Cahiers GUTenberg*, 44-45:3–53, Nov 2004.

[9] Yannis Haralambous and John Plaice. Methods for processing languages with Ω. In *Proceedings of the International Symposium on Multilingual Information Processing, Tsukuba 1997*, pages 115–128. ETL Japan, 1997.

[10] Donald E. Knuth. *TEX: The Program*, volume B of *Computers and Typesetting.* Addison-Wesley, Reading, MA, USA, 1986.

[11] Ferdinand de Saussure. *Cours de linguistique générale.* Payot & Rivages, 1916, facsimilé de 1995.