

Injecting Information into Atomic Units of Text

Yannis Haralambous*

Gábor Bella*

ABSTRACT

This paper presents a new approach to text processing, based on textemes. These are atomic text units generalising the concepts of character and glyph by merging them in a common data structure, together with an arbitrary number of user-defined properties. In the first part, we give a survey of the notions of character and glyph and their relation with Natural Language Processing models, some visual text representation issues and strategies adopted by file formats (SVG, PDF, DVI) and software (Uniscribe, Pango). In the second part we show applications of textemes in various text processing issues: ligatures, variant glyphs and other OpenType-related properties, hyphenation, color and other presentation attributes, Arabic form and morphology, CJK spacing, metadata, etc. Finally we describe how the Omega typesetting system implements texteme processing as an example of a generalised approach to input character stream parsing, internal representation of text, and modular typographic transformations. In the data flow from input to output, whether in memory or through serializations in auxiliary data files, textemes progressively accumulate information that is used by Omega's paragraph builder engine and included in the output DVI file. We show how this additional information increases efficiency of conversions to other file formats such as PDF or SVG. We conclude this paper by presenting interesting potential applications of texteme methods in document engineering.

Categories and Subject Descriptors

I.7.2 [Document and Text Processing]: Document Preparation—*Format and notation, Languages and systems, Photocomposition/typesetting, Markup languages, Desktop publishing, T_EX*; J.5 [Computer Applications]: Arts and Humanities—*Linguistics*

*Département Informatique, ENST Bretagne, CS 83818, 29238 Brest, France

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DocEng '05, November 2–4, 2005, Bristol, United Kingdom.
Copyright 2005 ACM 1-59593-240-2/05/0011 ...\$5.00.

General Terms

Theory, Standardization

Keywords

texteme, multilingual typesetting, Omega, character, glyph, Unicode, OpenType, PDF, SVG

1. INTRODUCTION

In the history of mankind the act of writing has always been considered as an activity producing *visual* results, namely text. The computer has brought a more abstract layer to it, by storing and transmitting textual data. The atomic unit of this abstract representation of text, as defined in the Unicode standard, is called a *character*. And indeed, characters prove to be useful for obtaining alternative (non-visual) representations of text such as Braille, speech synthesis, etc. The visual representation of a character is called a *glyph*. Displaying textual contents, whether on screen or on paper, involves translating characters into glyphs, a non-trivial operation for many writing systems. Going in the opposite direction (from glyphs to characters) is known as OCR when done by a machine, or as reading when done by a human.

The industry trend of the last few years has been to use characters for most of the text processing and to limit glyph issues to the last stage, namely *rendering*. At that level, character to glyph translation is handled by increasingly “intelligent” (cf. OpenType and AAT technologies) fonts. At the same time, for the opposite direction—restoring the original character stream from a rendered electronic document output for operations such as searching, indexing, or copy-pasting—no general solution exists in today's popular document formats yet. Examples in this article will show that going from characters to glyphs and then coming back can be a surprisingly complicated process.

In this paper we propose a new approach to the character vs. glyph issue: we merge them into a single atomic text unit, together with a number of additional properties related to language, presentation, or other types of metadata. We call such atomic units, *textemes*¹. A texteme can carry both a character and a glyph, but in some cases (such as in Indic or South-East Asian scripts where glyphs are not rendered in logical order) may very well be character-less or glyph-less, or both.

¹Or *micro-textemes* to avoid confusion with the concept of texteme in rhetoric.

Before introducing textemes let us recall some basic visual text processing related issues.

2. VISUAL REPRESENTATION ISSUES

When representing text visually, and furthermore “typographically” (that is, in accordance with a certain number of traditional conventions whose goal is to optimize readability), we deal with a number of important document engineering issues, such as hyphenation, kerning, and ligatures.

2.1 Hyphenation

Hyphenation rules vary greatly between languages: US English hyphenation is morpheme-influenced, and then syllable-oriented inside morphemes. In German, hyphenation is, in order of priority, first applied between word components. To hyphenate “Hühner-ei” but “Schweine-rei” one has to analyze the former as the plural form of noun “Huhn” + noun “Ei”, while the latter is noun “Schwein” + suffix “-rei”. In modern Greek, hyphenation is purely and shamelessly syllabic, totally ignoring morphological aspects (and, in particular, word component boundaries).

It follows from these considerations that, often, hyphenation faces the same ambiguity problems as text-to-speech conversion. Disambiguation needs several steps of linguistic processing: morphological, syntactical and semantic analysis. Once obtained, the list of potential hyphenation locations in a word is valuable information and should ideally accompany the text data (to avoid unnecessary re-calculation).

To store this information, Unicode provides the character U+00ad SOFT HYPHEN. Use of this character is suboptimal because, being a character in its own right, it interferes with regular “alphabetic” characters and can potentially hinder operations like kerning and ligaturing, especially when hyphenation points occur in the middle of a ligature. There are also non-standard hyphenations in German, Dutch, Hungarian, Greek and other languages; these are not covered by using solely character U+00ad. For example, in German the ‘ck’ letter digraph is hyphenated as ‘k-k’ (as in *backen* hyphenated as *bak-ken*).

2.2 Kerning

Kerning is a purely visual issue: because of their shape, some glyphs need to be brought closer together or farther apart. It is an important legibility factor: letters too far apart can be falsely interpreted as word boundaries, letters too close can be taken as a single glyph. The main technical problem is storage of kerning pair data: in current multilingual fonts, there are thousands of glyphs, and this can lead to tens of thousands of potential kerning pairs, some of which may be very rare. To circumvent this problem there are techniques for on-the-fly generation of kerning pairs and buffering of the most frequent ones.

In static presentation file formats such as PostScript, PDF and DVI, a kern becomes a shift, undistinguishable from shifts produced by interword spaces. It is inserted between glyphs, making identification of character strings more difficult. In formats such as RTF, (X)HTML, SVG there is no way to insert explicit kerning: it is up to the operating system to retrieve the necessary data from the font and render the text accordingly.

2.3 Ligaturing

From a visual point of view, ligatures are an extension of the kerning mechanism: when shifting glyphs is not sufficient, a new glyph is drawn, graphically combining the original ones. The typical example is ligature ‘fi’ obtained as a combination of ‘f’ and ‘i’. From a technical point of view, ligatures are quite different from kerning pairs since they involve a new glyph replacing the two or more original ones. Usually the new glyph has no matching Unicode character, unlike the original ones. Nevertheless, in document formats with interactive properties, ligature glyphs have to provide a link to the original glyphs and the characters behind them, to enable text searching and indexation.

Ligatures roughly belong to two classes: *esthetic* (like ‘fi’) and *grammatical* ones like the French ‘œ’ which, according to French grammar, appears whenever we have a pair of letters ‘oe’—this rule being subject to exceptions (like *moelleux*). However, linguistic factors are involved even for esthetic ligatures: for example, German morphological boundaries will break ligatures, as in *Auflage* (where the ligature ‘fi’ is broken: compare with *Abflug*). A more global example: ‘f-ligatures’ are prohibited in Turkish, where ‘fi’ is ambiguous (it can represent ‘fi’ or ‘fi’).

2.4 Combined Phenomena

Hyphenation can occur between kerned letters or in the midst of a ligature. In fact more than one “components” of a ligature glyph can be potential hyphenation points. When a ligature is broken, new ligatures can occur (for example, when breaking ‘affix’ into ‘af-fix’, a new ‘fi’ ligature appears). In other cases there can be kerning with the hyphen glyph or use of an alternate glyph at the beginning of the new line.

Phenomena like kerning and ligaturing change the total width of words, and this can affect the choice of optimal hyphenation point: to break a line L into parts L' and L'' , one needs to calculate the effective width of L' followed by a hyphen and it is not necessarily true that $\text{width}(L' + \text{hyphen}) = \text{width}(L') + \text{width}(\text{hyphen})$. Hence, when searching for optimal hyphenation points, the system must apply anew the font’s ligature and kerning rules on parts of the word before and after the hyphen.

3. ESTABLISHED STRATEGIES

3.1 SVG, PDF, DVI

SVG is currently the system that most elegantly deals with characters and glyphs. Separation of textual and visual data strictly follows the XML principle: text, and only text, is encoded as parseable character data. As this is the default for XML, text is encoded in Unicode characters.

To identify or describe glyphs, SVG uses the notion of “current font,” attached to each `text` or `tspan` element. This information 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 a `font` element containing `glyph` elements. The latter has a `unicode` attribute which is the Unicode character sequence that is rendered by this glyph. The contents of the `glyph` element can be arbitrary SVG code.

Another way to obtain a specific glyph is through element `altGlyph`, the `PCDATA` content of which is the corresponding Unicode character sequence. It takes a number of

attributes, among which we have an XLink reference to the corresponding glyph element inside the font and the absolute coordinates of its origin. One can also use the usual SVG attributes (style, color, opacity, conditionality, etc.).

The `altGlyph` element comes quite close to the notion of *texteme*: it provides both a character (in fact, a character sequence), a glyph, and some additional properties expressed by attributes, although the latter are not entirely user-definable as in the case of *texteme* properties.

PDF is predominantly oriented towards visual presentation. As in PostScript, text is stored using “character codes” (which are actually glyph indexes), mapped to (single) Unicode characters through a special table (`ToUnicode`) in the font entry. This method has several disadvantages: mapping between glyphs and characters is hard-coded to the font, and only one-to-one correspondences are possible. Nevertheless, PDF has a separate mechanism for mapping strings of glyphs to strings of characters: the `ActualText` operator [10, p. 872]. The example given is that of German ‘ck’ hyphenation:

```
(ba) Tj /Span <</ActualText (c) >>
BDC (k-) Tj EMC (ken) ’
```

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-.” Regarding interactivity, this means that when a user selects the word *bak-ken* what is copied in the clipboard is *backen*. Nevertheless this approach has a pitfall: the two strings that are mapped to each other become indivisible objects. This is suboptimal for longer strings where the user may want to select only part of the logical or visual string. In addition, only the “logical” part is available for copying, searching, and indexing.

DVI handles glyph indices only. Even if we provide \TeX or Omega with Unicode characters, they are immediately replaced by “character nodes”: (f, g) pairs where f is a font identifier and g a glyph index. Text as represented in a DVI file cannot be directly mapped to Unicode characters, unless we know the exact encoding of each font.

3.2 Uniscribe and Pango

In recent Windows systems typesetting services are provided by system-level libraries of which the most important is *Uniscribe* and by higher-level solutions such as the *RichEdit control*. *Uniscribe* is a low-level shaping engine that produces formatted output for several writing systems, provided that appropriate OpenType fonts are available. There is a free, cross-platform alternative to *Uniscribe* called *Pango*, mainly used on Unix systems.

Both *Pango* and *Uniscribe* assume that the input text is a string of Unicode characters accompanied by *properties* and their *ranges* (in Windows terminology). A range is a substring with a given set of properties. For example, the 9-character long string ‘THIS TEXT’ will likely have two range objects attached: the range 0-5 with the bold property set to true and the range 6-8 with bold set to false. Individual (script, font, style etc.) and paragraph-level (justification, line spacing etc.) properties, although implemented independently, are both specified as ranges. The set of available properties is hardcoded into the two libraries and is thus fixed.

While *Uniscribe* needs to be called with the range objects already defined for the input character string, *Pango*

is also able to parse tagged input, that is, character strings with `` and similar elements. This is a very simplistic but effective way (even if it is *Pango*-specific, despite the HTML-ish style) to add supplementary information to the input text, be it formatting instructions, semantic or other information. It is not possible, however, to process user-defined properties inside *Pango* without modifying the library itself. *Pango* lets unknown properties pass through, untouched.

Another difference between the two libraries is that *Pango* allows overlapping property ranges while *Uniscribe* seems to assume that the client always defines disjointed ranges.

Concerning the character-glyph duality, both *Pango* and *Uniscribe* shaping engines are aware of the fact that conversion of characters to glyphs can be an $N:M$ mapping. Their solution is to include an additional pointer array along with the output glyph string which indicates the exact character-to-glyph correspondences. (For example, the second, third, and fourth characters of the input string are mapped to the third and fourth glyphs of the output string.)

In conclusion, both libraries provide sufficient information to keep the possibility of bijection between character and glyph data. It is nonetheless up to the client to take advantage of this information and to solve the problem of outputting them into the formatted document, a process far from evident, as shown by the examples of PDF and SVG. Finally, as far as text properties are concerned, both libraries are bound by their own abilities: apart from using the predefined properties they offer, there is no way for users to have any influence on the typesetting process by adding their own properties and letting them define their own transformations.

4. TEXTEMES AND THEIR APPLICATIONS

4.1 Definitions

A *texteme* is a set $\{c, p_1 = v_1, \dots, p_n = v_n, g\}$ where c is a Unicode character codepoint, g a glyph index (or pointer to a glyph description), and p_i ($0 \leq i \leq n$, for arbitrary n) a list of named properties taking values v_i . Property values can be numbers, character strings or pointers to other *textemes*.

In fact, c and g can also be considered as simple *texteme* properties (called “character” and “glyph”).

Property values (including character and glyph) can be selectively locked to prevent modification in subsequent steps of text processing.

A *texteme pattern* is a pattern based on any combination of character, glyph, and/or property values.

In this paper we will use the following generic notation for

textemes:

| | |
|-----------|---|
| c=062C | ☺ |
| form=1 | |
| color=red | |
| g=⚡, 18 | |

 where the first row is the Unicode character codepoint, the last row the glyph (given either by the glyph image or by its numeric index in the current font) and the intermediate rows are properties given by key/value pairs.

4.2 Principal Features of Textemes and Properties

Merging characters and glyphs (and other properties) in a single atomic unit allows multiple level processing:

- traditional text processing tools can be applied to the character part of textemes leaving other parts unchanged;
- glyph-related processing such as OpenType tables can be applied to glyph parts, leaving characters unchanged;
- more sophisticated tools can match and act upon various parts of textemes.

Texteme properties are identified by their names. Unique identification can be achieved by using the idea of namespaces, borrowed from XML.

Property values are mostly independent, so that a texteme can be considered as a finite-dimensional vector in an infinite-dimensional vector space.

These principles lead us to the conclusion that text processing in general, and typesetting in particular can now be viewed as a *progressive accumulation of texteme properties*: one may start with textemes containing almost exclusively characters, and progressively fill in other properties such as glyph, coordinates on the page, colour, additional semantics, etc. The dividing line between “raw” and “formatted” text will eventually fade, the only difference being in the number of properties attached to textemes.

4.3 Textemes in Input and Output Documents

So far we have presented the texteme concept as a means to improve text processing. However, textemes may not only be used outside of typesetting engines but also in their input and output documents. More specifically, by “input document” we mean raw-text based files interspersed with structural, semantic, and/or formatting information. “Output documents” are the formatted output of the typesetting engine.

Input documents consist of *texteme streams* where most textemes contain almost exclusively a single “Unicode” property. *Almost* since the the document author may sometimes want to include additional properties into the text, wishing to use alternate glyphs (e.g., swashes, oldstyle numerals) for certain characters or to forbid hyphenation at some point and so on. (This, however, requires a text editor capable of handling texteme-based documents.²) A texteme-based typesetting engine would then read this input texteme stream, format the text by adding new properties, and finally generate an output document file based on these enriched textemes. By consequence, the formatted output document would contain texteme strings very similar to the original ones from the input file but extended by formatting properties.

Texteme-based documents are only one possible solution to add extra information to raw text. Existing and well-known methods such as *text markup* (HTML, SVG, T_EX etc.) or *Unicode control characters* (in some special cases) are already widely used to obtain similar results. In the following section we will discuss whether textemes can be

better suited for enriching raw text than text markup or Unicode control characters.

4.4 Texteme Strings, Unicode Control Characters, or Higher-Order Markup?

Textemes can be considered as a bridge between characters and higher-order markup. Indeed, the boundary between the latter two (in more prosaic terms: between the Unicode world and the XML one) is unclear and usage may vary: sometimes the same information may be either encoded by special Unicode characters or represented by markup. As shown by the following example, while both approaches are used today, neither of them is able to provide an optimal solution for certain text processing issues.

Ligatures between subwords, like between ‘f’ and ‘l’ in the German word *Auflage*, are not allowed in German typography. Independently of how subword boundaries are identified (by automatic lexical analysis or author input), current document engineering technologies provide two possibilities to include this information into the input document: either by inserting markup (like the {} combination in T_EX) or by a special 200C ZERO WIDTH NON JOINER Unicode character. The problem with both approaches is that they introduce “artefacts” into the document that disrupt natural text flow. Similar situations are shown in section 4.6.5, such as the one where semantical disambiguation is needed to decide if a period is used to end a phrase or an abbreviation.

Text-based markup has a great advantage of being human-readable which is an important criterion of the openness of document formats. Unfortunately, markup can also be sub-optimal for the very same reason: it increases document size excessively. Unicode control characters, on the other hand, while certainly unbeatable from the point of view of information quantity carried per character, are also a difficult concept to grasp for users because they affect the text while at the same time remain “invisible”. Moreover, and perhaps more importantly, both solutions require traditional text-processing operations (such as searching or regular expression pattern matching) to be aware of such artefacts in the textual content and be ready to include or exclude them when necessary. In the case of our example, a search operation should be able to simply find the word *Auflage* independently of the markup or control characters inside it, but *also* to eventually identify places in the text where such separator elements are used.

A lot of these problems can be solved by using textemes. Otherwise verbose markup can be limited to the strict minimum. Low-level text units such as words can be encoded as texteme strings interrupted neither by markup nor by control characters. Regular expression syntax can be extended to allow property-based pattern matching.

Finally, a further constraint on using markup is that, according to XML specification, markup cannot be used *inside* markup. Indeed, only character data are allowed in attribute values or in tag names. This restriction is also lifted by using textemes: XML parsers, validators, transformers, and other XML-related tools still act on characters (that is: on the character part of textemes) as stated in XML specification, independently of the additional information contained in textemes.

²An extended, texteme-compatible version of the Java-based open source text editor *jEdit* is currently being developed by students at ENST Bretagne.

4.5 Performance Issues

Plain Unicode-based text usually does not occupy more than two bytes per character. Textemes, on the other hand, are more complicated data structures that may take up several times more space (depending on the number and type of properties stored per texteme). How does using textemes instead of simple characters increase memory and processor use?

The answer to this question depends on where and how textemes are being used.

In a conservative scenario, textemes would solely be created and used inside the typesetting engine. This already improves and simplifies several text processing tasks.³ Inside the processing engine, memory use can be significantly reduced by indexing frequently-used property values and by other optimisation techniques. Moreover, a typesetting engine usually does not need to keep a whole document in memory all the time but only the part it is working on, which is not more than one or two pages. The T_EX system, as well as Omega, T_EX’s multilingual and Unicode-enabled successor⁴ both work according to this principle. Texteme support is under development for the Omega system (cf. section 5), and though work is not finished yet, it can be estimated that the memory used up by a two-page long portion of an ordinary texteme-based document is well below a Megabyte, and this without any optimisation trickeries.

If textemes are not only used inside the text processor but also to encode input and output documents, optimal serialisation of data structures holding textemes and properties becomes much more important. Even with mass storage becoming less and less expensive, documents should not grow to several times their original size. Fortunately, input document textemes are rather sparsely populated by properties: in most cases, only the *character* property is used (i.e., the Unicode value itself). Semantic or formatting information added explicitly by the user do not occur often in the input document; as a consequence, input document size is rather easy to reduce by clever design of the file format.

In formatted output documents, textemes hold a lot more information including glyph indices, coordinates etc. As a result, designing a storage-efficient output format is a more delicate issue. Apart from techniques mentioned earlier (indexing, compression), a careful study of *property recycling* becomes necessary: properties that can easily be recalculated later may be disposed of (e.g., metrics, given that font files are available) whereas others may need to be kept (e.g., mapping between character codes and glyph indices, user-introduced semantical information).

Let us conclude this section by saying that at the moment of writing the present article, texteme-based document formats remain a question open to research.

4.6 Examples of Application

Having stated the basic properties of textemes, here are some examples that will better illustrate their applications.

4.6.1 Arabic Form and Morphology


Arabic letters can take one to four different contextual forms. Unicode does not encode Arabic forms.⁵ Operating


³We will illustrate our point with examples in section 5.



⁴Developed by the John Plaice and the first author.

⁵Except for the “Arabic Presentation Forms” tables, which have been included in the standard for “legacy reasons” and are im-

systems and software use contextual analysis algorithms to render Arabic character strings: the same calculations are repeated whenever a string has to be rendered. Textemes allow storage of the Arabic form information to avoid redundant contextual analysis.

Furthermore, in the case of exceptions to regular contextual analysis, the means to obtain a glyph with a specific form is by preceding or following its character by characters 200C ZERO WIDTH NON JOINER and 200D ZERO WIDTH JOINER. In this way, to obtain the visual representation  (an initial *jeem* letter, which is used in dictionaries [6] as an abbreviation of جمع = plural) one has to use the character string 062C ARABIC LETTER JEEM 200D ZERO WIDTH JOINER where the second character acts as a fake Arabic letter. Using a two character string here is unnatural since the status of the zero width joiner in algorithms such as segmentation, searching, indexing, etc. is not clear.⁶ Using

textemes,  is encoded as the single unit

| | |
|--------|---|
| c=062C |  |
| form=1 | |
| g= |  |

.

Textemes can also be used to inject morphological information into Arabic letters. In this way one could mark Semitic roots and affixes without introducing additional characters. For example, the fact that in *al^ukit^uabu* underlined letters belong to the definite article and bold ones to the Semitic root **ktb** can be expressed by texteme properties:

| | | | | | | | |
|----------------------------------|----------------------------------|----------------------------------|------------------------|----------------------------------|-------------------------|----------------------------------|------------------------|
| c=0627 art=1 form=0 g=ا | c=0644 art=1 form=1 g=ب | c=0643 sem=1 form=2 g=ك | c=0650 vow=1 g=ـ | c=062A sem=2 form=2 g=ا | c=0627 form=3 g=ل | c=0626 sem=3 form=0 g=م | c=064F vow=1 g=ن |
|----------------------------------|----------------------------------|----------------------------------|------------------------|----------------------------------|-------------------------|----------------------------------|------------------------|

where **art** is used for the definite article, **sem** for the order in the Semitic root, **vow** for short vowels (Unicode combining characters).

4.6.2 OpenType Features

OpenType tables GSUB and GPOS provide information on various glyph transformations, based on user activated “features,” each feature attempting “lookups” (pattern matching on the glyph string), and applying a series of glyph substitutions and/or positioning for each matched lookup. Textemes can contain both the names of activated properties and the result of glyph transformations. Let us review briefly some types of OpenType transformations [5, p. 746–785]:

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

a small cap one:

| |
|------|
| c=a |
| sc=1 |
| g=a |

 →

| |
|------|
| c=a |
| sc=1 |
| g=A |

 ;

- *multiple substitution*: a glyph is replaced by more than one glyphs. For example, we may want to replace the ideographic square **klz** by the glyph string “kHz” (additional character-less textemes are generated):

| |
|------------|
| c=3391 klz |
| g=klz |

 →

| |
|------------|
| c=3391 klz |
| g=k |

| |
|-----|
| c= |
| g=H |

| |
|-----|
| c= |
| g=z |

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

texteme property:


| |
|-------|
| c=& |
| alt=3 |
| g=& |

 →

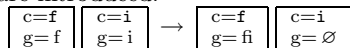
| |
|-------|
| c=& |
| alt=3 |
| g=8 |

 ;

explicitly ostracized by the Unicode consortium.

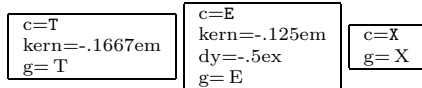
⁶And furthermore, because of the restrictions on XML names, there can be no  XML element tag.

- *ligature substitution*: the ordinary ligature. Glyph-less textemes are introduced:



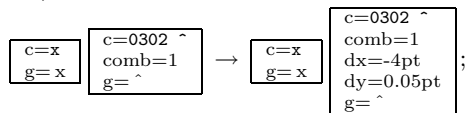
- *single adjustment*: position adjustment of a glyph. We introduce texteme properties **dx** and **dy** for horizontal and vertical glyph offset (without changing the glyph's advance vector).

A typical example of such a transformation is the $\text{T}_{\text{E}}\text{X}$ logo, which becomes a plain texteme string:



(see below for the **kern** property);

- *pair adjustment*: adjustment of a pattern of two glyphs. Kerning is the most common case of pair adjustment. We define texteme properties **kern** and **vkern** for horizontal and vertical vectors, algebraically added to the advance vector;
- *cursive attachment*: defining marks (points in the glyph's coordinate space) on each side of a glyph, and typesetting in such a way that the right mark of glyph n is identified with the left mark of glyph $n+1$. This is an alternative to kerning (based not on kerning pairs, but on generic marks) but after calculations we still obtain **kern** and **vkern** properties as results;
- *mark to base attachment*: the principle is the same as for cursive attachment, but this time the goal is to connect a "base" to an "attachment." Marks are defined on strategical positions around the letter (roughly corresponding to Unicode combining classes) and around the accent. When the glyph of a combining character follows that of a base character, the appropriate pair of marks is identified, leading to specific values of **dx** and **dy** properties. Here is an example (the hypothetical letter 'x̂'):

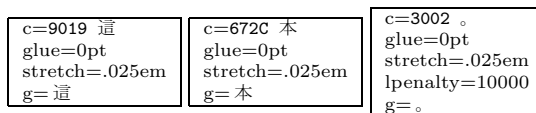


- *mark to mark attachment*: same principle but attaching the glyph of a combining character to that of another combining character;
- *mark to ligature attachment*: same principle but defining individual marks for ligature components. This is rarely encountered in Latin fonts, but becomes crucial in Arabic fonts with many ligatures (since short vowels and other diacritics are considered as Unicode combining characters).

4.6.3 CJK Spacing

Constraints imposed on the typesetting of Chinese ideographs imply the use of small equidistributed amounts of kerning on a given line to avoid the subsequent line starting with a punctuation mark or with a closing delimiter. Calculation of the amount of kerning needed can follow the $\text{T}_{\text{E}}\text{X}$ model of penalty and glue: *penalty* is a measure of optimality of line breaking at a given position and *glue* is a range (expressed by an ideal value, and maximum stretch and shrink) of permitted interglyph kerning, potentially including infinite values of various orders. We extend this model by using penalty values for both sides of the glyph. Here is an example of glue and left penalty values for the

string “這本.”:

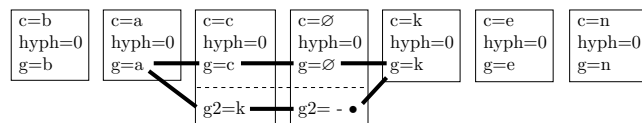


4.6.4 Hyphenation

In most cases indicating which letter in a word is a potential hyphenation point is sufficient for paragraph engines. Storing this information in text can be practical because hyphenating is a complex process requiring knowledge of current language and hyphenation conventions, and NLP resources such as a morphological analyzer for cases like the word “record” which hyphenates differently depending on whether it is a verb (to re-cord) or a noun (a rec-ord). Unicode provides character `00AD SOFT HYPHEN` to encode hyphenation points, but using glyphless characters inside strings can hinder other operations such as kerning, ligaturing, etc.

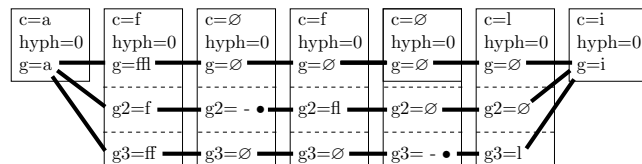
We introduce a texteme property for hyphenation points. In most languages this will be a boolean, but in some cases ponderation is useful, like in German where the rule is to attempt hyphenation first between word components, then between syllables of the last component and finally between syllables of other components: *Wahr₁schein₃lich₃keits₁the₂o₂rie*. In this case the value of the hyphenation property can be an integer indicating priority.

To store non-standard hyphenation phenomena in textemes we introduce a new concept: a *bifurcation* is a graph structure on the level of glyph parts. We illustrate bifurcation by a thick stroke in the following example (the German word *backen*):



where the bullet indicates a mandatory line break. The rendering engine will have to choose a path through this graph: either the word is not broken and the path goes through glyphs ‘b’, ‘a’, ‘c’, (empty glyph), ‘k’, ‘e’, ‘n’ or it is broken and the glyphs are ‘b’, ‘a’, ‘k’, ‘-’, (line break), ‘k’, ‘e’, ‘n’. Notice that the texteme carrying the hyphen has an empty character part and that none of the textemes is a potential hyphenation point (property **hyph**) any longer.

The situation is more complex when a hyphenation point happens to fall inside a ligature: in that case the bifurcation graph must contain one path for each hyphenation point, providing all new ligatures occurring in case of hyphenation. In the figure below we see ligature ‘fff’ in the hypothetical case where hyphenation is allowed on both sides of the second ‘f’:



The rendering engine's task is then to choose between one of the paths, depending on which line breaking is optimal.

4.6.5 Additional Semantics

Most of our texteme application examples deal with visual representation issues. In some cases a Unicode character has several semantics which we may want to disambiguate in order to optimize text processing. For example, the character 002E FULL STOP stands both for the end-of-sentence period, for the abbreviation mark, or the two. Distinction between these semantics is important for visual representation since according to the typesetting rules of English the space after an end-of-sentence is larger than the usual interword space, to optimize visual detection of sentence boundaries. Establishing algorithmically whether a period is the end of a sentence is quite a complex task involving morphosyntactic, semantic and even pragmatic analysis: in “I miss the strawberries, etc. Julia brought” it is clear that the period is not an end of sentence because “Julia brought” lacks a direct object. But in “I miss the strawberries, etc. Julia has eaten,” analysis must go one step further because of possible intransitivity of verb “to eat.”

After having applied heavy linguistic machinery to classify period characters one obviously needs to store this information in the document. The TEI DTD [9] uses entities for this purpose, while T_EX requires insertion of a control sequence. To avoid markup one can store this information inside a texteme.

The same technique can be applied whenever a semantic distinction finer than Unicode character description is needed: for example to distinguish between “intended as opening English double quote and as closing German double quote (*Gänsefüßchen*), or between ’ used as closing English single quote or as French apostrophe. These distinctions may or may not have an effect on visual representation.

A typical example is *case*: while Unicode defines three cases (lower, upper and title case), other cases may prove useful as well. For example, there is *mandatory upper case*: in “Paris” (capital of France) the ‘P’ is mandatory upper case, while in “Paris” (plural of word “pari,” at sentence beginning) it is “accidental.” In German abbreviation “GmbH,” as well as in units such as “mW” (milli-Watt) the lower case is mandatory. Finally there is also *inverted case* which is contextually defined by the surrounding letters: in the German neologism “StudentInnen” the ‘I’ is in inverted case: the word becomes “STUDENTINNEN” in the all-caps context. Case is a typical candidate for a texteme property since storing case information by higher level markup would inflate the document and hinder text-related operations.

5. TEXTEMES IN THE OMEGA SYSTEM

The examples in the previous section demonstrate that the texteme concept proves to be useful in a variety of non-trivial cases of text processing. However, we have yet to examine the other side of the coin, namely practical issues of building a typesetting engine based on textemes. For experimentation, our platform of choice is the Omega typesetting system, created as an extension of T_EX by John Plaice and the first author [4, 7, 8]. Texteme support in Omega is currently under development; in the following we are going to present how the texteme concept can be integrated into the system and how it improves document processing.

5.1 Introduction to T_EX and Omega

For the reader unfamiliar with Donald Knuth’s T_EX, a document written in the T_EX programming language is a

mixture of plain text and markup where markup mostly represents formatting commands and parameters. T_EX starts by parsing input and interpreting commands (this process is called *tokenization*), then converts the resulting tokens into *nodes* and builds *node lists*. A token is either a pair (f, g) (font identifier, glyph index) or a special value that represents a command code. The token stream is further converted by T_EX into node lists. A node is the basic building block of formatted text inside T_EX’s memory: it can be a glyph, a kerning offset between two glyphs, a hyphenation point, a ligature, a penalty, etc. A node list is a text block—such as a paragraph—described as a linked list of nodes. Node lists may contain other node lists (represented by special ‘insertion’ nodes), like in the case of a paragraph (logically) containing a footnote.

Omega is an extension of Knuth’s T_EX with an emphasis on multilingual typesetting and Unicode-compliance. It is Unicode-based and introduces multiple writing directions as well as the ability to *filter* documents by extensible and modular *translation processes* (called OTPs). An OTP is a finite-state automaton given by a set of rules transforming streams of tokens. Transformations involved in typesetting are writing system- and language-dependent, so that each one will typically have its own set of OTPs, ranging from transcoders (such as from various 8-bit encodings to Unicode) through ligaturing and contextual analysis to diacritical mark positioning.

Typesetting text through a series of filters is common in today’s layout systems: intelligent font technologies such as OpenType, Apple’s AAT, or SIL Graphite all define some kind of finite state automata (or similar) to describe transformations. What makes OTPs unique is their extensibility: the user has complete control over which OTP filter is active at which point of the typesetting process.

5.2 Implementing Textemes inside Omega

In our model, typesetting is about accumulating and transforming properties inside textemes. This is exactly what the Omega system does: when parsing the input text file, textemes are created instead of character tokens. Unless the input file has been prepared with a texteme-compliant text editor, these textemes will initially contain very few properties: only the *Unicode* and the *font* properties will be filled (plus some formatting properties requested by the user, like *small caps* or *superscript*). Then, progressively, textemes accumulate more and more properties: glyph indices from the font, potential hyphenation points provided by a hyphenation engine OTP. Textemes become the new memory elements for text, replacing character tokens and nodes at the same time. Moreover, several of T_EX’s original node types disappear: horizontal and vertical kerning and offset, hyphenation, penalty, glue (stretchable/shrinkable space for justification), color etc. nodes can all be included in textemes as properties. What was formerly a list of abstract T_EX-specific structures (nodes) now becomes a simple texteme string.

5.3 Textemes in Filters

As explained above, Omega does most of its text transformations using filters called OTPs. There are two kinds: *internal OTPs* are finite state automata written in a syntax close to that of *lex*, while *external OTPs* are separate binary executables. The advantage of the latter is that trans-

formation tools of arbitrary complexity can easily be called from inside Omega provided that they comply with its input-output format.

OTPs contain *rewrite rules* that apply regular expressions to the token stream. Before the implementation of texteme support, rules had access only to character token streams and in this, very limited possibilities apart from contextual character replacements. Moreover, due to the T_EX heritage, there was no clear distinction between characters and glyphs: rules only applied to character values, and fonts had to be engineered to maintain the proper character-glyph mapping.

Filters can be used in much more diverse ways when allowed to operate on textemes. The OTP syntax is extended to allow arbitrary access to texteme properties. To give a comparison, without textemes one would use (in three subsequent OTPs):

```
<INITIAL> @"0644 => @"FEDF ;
@"0645 => @"FEE3 ;
```

then:

```
@FEDF @FEE4 => @FCCC ;
```

and finally:

```
@FCCC => @"0154 ;
```

to first replace the *lam* and *meem* Arabic Unicode characters by their initial and medial forms (using the Arabic Presentation Forms range of Unicode), then to replace an initial *lam* + medial *meem* string by an initial *lam-meem* ligature (Unicode @"FCCC in Presentation Forms), and finally to replace the Unicode character by the glyph index.

There are at least three problems with this method: first, as already mentioned, Presentation Forms are not the optimal way to encode Arabic. Secondly, these hybrid characters/glyphs are necessarily replaced in the final step by a glyph index, resulting in loss of reference to the original character string. And finally, in the very frequent case where a short vowel has to be placed on the *lam* there is no longer any Presentation Form coverage. The only Unicode-compliant workaround would be a mapping through the Private Use Area, once again resulting in loss of reference to the original characters.

Texteme-based OTPs apply regular expressions to any texteme property and may add, modify, lock, unlock and remove them. The composition of the *lam-meem* ligature becomes much more elegant:

```
<INITIAL> [c=0644 form=1 g=*] => [c=0644 form=1 g=*] ;
[c=0645 form=2 g=*] => [c=0645 form=2 g=*] ;
```

then:

```
[c=0644 form=1 g=*] => [c=0644 form=1 g=J] ;
[c=0645 form=2 g=*] => [c=0645 form=2 g=meem] ;
```

and finally:

```
[c=0644 form=1 g=J] [c=0645 form=2 g=meem] => [c=0644 form=1 g=lam] [c=0645 form=2 g=meem] ;
```

First, a contextual analyzer adds the initial form property to the *lam* texteme and the medial form to *meem*. The second rule could result from an OpenType lookup that inserts glyph indices into the textemes. Finally, another OpenType-based rule replaces the lam glyph by a *lam-meem* ligature and removes the *meem* glyph. The resulting texteme string includes both the original characters and the ligature glyph.

One of the main advantages of textemes is that the set of property types is open: users may add self-defined properties to their input text (provided that there is a way to define textemes in the input file) that can pass through the typesetting engine and appear in the output document without alteration. By using user-definable OTP filters, the Omega system is able to make transformations on these properties. To take the example of Arabic morphology from section 4.6.1, it is easy to imagine an OTP that marks Semitic roots of words in a different colour (supposing that the Semitic root

```
is a user-defined property): [c=* sem > 0] => [c=* sem=\1 color=red g=*] ;
```

5.4 OpenType Support in Omega

A typesetting engine compliant with the OpenType font format needs to be consistent with OpenType's philosophy: this involves Unicode support and a clear distinction between characters and glyphs since input text is character-based while all OpenType transformations are applied on glyph strings. Moreover, it has to be aware of the particularities of every writing system and language supported and be able to preprocess character strings accordingly. Proper use of OpenType is not possible without this knowledge: OpenType has a declarative nature that usually says *what* to do but not *how* to do it.

The texteme-based version of Omega copes with these requirements. Character and glyph data are always separated inside textemes. Script-specific preprocessing such as passage from special transcription schemes to Unicode or reordering of the input character stream (from logical to processing order) is done by OTPs. OpenType's two most important transformation tables, GSUB (glyph substitution) and GPOS (glyph positioning) are automatically converted into OTPs that act on the glyph index, *dx/dy* and/or *kern/vkern* texteme properties. The user decides at what processing stage the GSUB- and GPOS-derived OTPs are applied to the texteme stream. Usually—but not necessarily—this happens at the last stage before paragraph building.

This also means that it becomes possible to intervene in OpenType's transformations before, between, and even after GSUB and GPOS lookups.

One case where such an approach may be useful is mark positioning in Arabic or Hebrew, a task which is non trivial [3] due to the large number of possible base letter and mark combinations, ligatures etc. For such fonts, one would first apply standard OpenType GSUB transformations, then circumvent the font's own positioning methods and apply external algorithms as OTPs. These OTPs may access glyph contours and use diverse automatic methods to calculate mark positions based on visual properties of the contour.

The resulting coordinates are then included in the `dx` and `dy` properties of mark textemes.

6. CONCLUSION

By clearly separating the various information strata of text (characters, glyphs, various kinds of properties), textemes allow more intelligent processing, and in particular, optimized typesetting. Thanks to textemes, tools that have traditionally been used for characters only (like regular expressions, natural language processing tools) or for glyphs only (like OpenType tables) can now be combined and extended, opening new horizons to text processing.

7. REFERENCES

- [1] Raph Levien, *Is the SVG working group about to choose shame and get war?*, <http://www.levien.com/svg/shame.html>.
- [2] Martin Dürst et al., *Character Model for the World Wide Web 1.0: Fundamentals*. W3C Recommendation 15 February 2005, <http://www.w3.org/TR/2005/REC-charmod-20050215/>.
- [3] Yannis Haralambous, *Tiqwah, a Typesetting System for Biblical Hebrew Based on T_EX*, Actes du Quatrième Colloque International *Bible et informatique, matériel et matière*, Amsterdam, 1994, pp. 445-470.

- [4] Yannis Haralambous, *Unicode, XML, TEI, Ω* , International Unicode Conference 16, Amsterdam, 2000, pp. b.7.1–b.7.23.
- [5] Yannis Haralambous, *Fontes & codages*, O'Reilly France, 2004.
- [6] *Mounged de poche*, Dar el-Machreq, Beyrouth, 1986.
- [7] John Plaice, Yannis Haralambous and Chris Rowley, *An Extensible Approach to High-Quality Multilingual Typesetting*, IEEE Research Issues in Data Engineering: Multi-lingual Information Management, 2003. RIDE-MLIM 2003, Hyderabad.
- [8] John Plaice, Yannis Haralambous, Paul Swoboda and Gábor Bella, *Moving Ω to an Object-Oriented Platform*, Springer Lecture Notes in Computer Science 3130, 2004, pp. 17–26.
- [9] *Text Encoding Initiative*, <http://www.tei-c.org.uk/Drafts/P4/C0.xml>.
- [10] *The Adobe PDF Reference, Fifth Edition, Version 1.6*, <http://partners.adobe.com/public/developer/en/pdf/PDFReference16.pdf>.