Ω, a TEX Extension Including Unicode and Featuring Lex-like Filtering Processes

Yannis Haralambous
Centre d'Études et de Recherche sur le Traitement Automatique des Langues
Institut National des Langues et Civilisations Orientales, Paris.
Private address: 187, rue Nationale, 59800 Lille, France.
`Yannis.Haralambous@univ-lille1.fr`

John Plaice
Département d'Informatique, Université Laval, Ste-Foy, Québec, Canada G1K 7P4
`plaice@ift.ulaval.ca`

Abstract

Ω consists of a series of extensions to TEX that improve its multilingual capabilities. It allows multiple input and output character sets, and will allow any number of internal encodings. Finite state automata can be defined, using a `flex`-like syntax, to pass from one coding to another.

In this paper both a technical introduction and a few applications of the current implementation of Ω are given. The applications concern typesetting problems that cannot be solved by TEX (consequently, by no other typesetting system known to the authors). They cover a wide range, going from calligraphic script fonts (Adobe Poetica), to plain Dutch/Portuguese/Turkish typesetting, to vowelized Arabic, fully diacriticized scholarly Greek, or decently kerned Khmer.

A few problems Ω *cannot* solve are mentioned, as challenges for future Ω versions.

The basic idea of Ω is to separate the three main components of a typesetting system:

User interface — Information interchange — Typography

These components are not clearly separated in TEX, and this causes many problems when languages other than English are dealt with. Here is an example of this situation, as pointed out in Vulis 1989: to typeset the Cyrillic character ц using the AMS macros and font family (Beeton 1985), one has to enter the ASCII codes `ts`. The fact of typing `ts` is part of TEXs user interface component. The fact of obtaining ц is part of the typography component of TEX. The information exchange component is neglected: a text encoded in KOI-8, or ISO 8859-5, cannot be directly processed using this scheme: either pre-processing, or active characters have to be used to convert it into the AMS transliteration. The problem that can arise is that the hyphenation algorithm—which in the case of English will naturally break an 'fi' ligature into 'f-i'—may break ц into т-с, which is unacceptable. This happens because hyphenation (which is a process belonging clearly to the typography component of TEX), is applied while the data is still encoded as in the user interface component: ц is not yet recognized as a single entity, TEX only knows about the pair of ASCII codes `ts`.

In Ω this problem is avoided; moreover, any information interchange encoding can be used for the text input. This happens because separate processes convert the information stream between the three components: in the user interface component, one may very well type `ts`, it will be converted into the ISO 10646/Unicode character `0x0446` (CYRILLIC SMALL LETTER TSE). Text encoded in any information interchange encoding, either 8-bit or 16-bit, will also be converted into ISO 10646/Unicode. Finally, before entering the typography component of Ω, ISO 10646/Unicode characters will be mapped to output font dependent positions, and only then the hyphenation algorithm will be applied.

This paper is divided in two sections: the first, derived from Plaice 1994, gives a technical introduction to Ω; the second, derived from Haralambous and Plaice 1994, describes some straightforward applications, mainly in the field of multilingual typesetting.

An Introduction to Ω

Encodings and recodings. If we abstract ourselves from the problems associated with layout, typesetting can be perceived as a process of converting a stream of characters into a stream of glyphs. This process can be straightforward or very complex. Probably the simplest case is English, where, in most cases, the input encoding and the font encoding are both ASCII; here, no conversion whatsoever need take place. At the other extreme, we might imagine a Latin transcription of Arabic that is to generate highly ligatured, fully vowelized Arabic text; here, the transliteration must be interpreted, then the appropriate form of each consonant must be chosen, then the ligatures and vowels have to be chosen — the process is much more complex.

TEX supposes that there are two basic encodings: the input encoding and the internal encoding, each of which uses a maximum of eight bits. The conversion from the input encoding to the internal encoding takes place through an array lookup (`xord`). An input character is read and converted according to the `xord` array. The font encoding is the same as the internal encoding, except of course for the fact that several characters can combine to form ligatures.

Suppose that one works in a heterogeneous environment and that one regularly receives files using several different encodings. In this case, one is faced with a problem, because the TEX conversion of input to internal encoding is hard-wired into the code. To change the input encoding, one actually has to change TEX's code, hardly an acceptable situation.

So how does one get around this problem? The first possibility is to use preprocessors, which might themselves be faulty, before actually calling TEX. The second is to use active characters. At the top of every file, certain characters are defined to be macros. However, this process is unreliable, since other macros might expect those characters to be ordinary letters.

Much more appropriate would be to have a command that states that the input encoding has changed and, on the fly, that TEX switches conversion process, maintaining the same internal coding — if we are still in the same document, we probably want to use the same font.

It would probably not be too much trouble to adapt TEX so that it could quickly switch from one one-octet character encoding to another one. However, there are now several multi-octet character sets: JIS, Shift-JIS and EUC in Japan, GB in China and KSC in Korea. Some of these are fixed-width stateless 16-bit codes, while others are variable-width codes with state. Also, now that the base plane of ISO 10646-1.2 (Unicode-1.1) has been defined, we have a 16-bit character set that can be used for most of the world's languages. However, for reasons of compatibility, we may often come across files in UTF format, where up to 32 bits can be stored in a variable width (1-6 octets) encoding, but for which ASCII bytes remain ASCII bytes. In other words, the conversion process from input encoding to internal encoding is not at all simple.

To complicate matters even further, it is not at all clear what should be the internal encoding. Should it be fixed, in which case the only reasonable possibility is ISO 10646/Unicode? Or should the internal coding itself be variable? If the internal coding is fixed, that will mean that in most cases a conversion from internal encoding to font encoding will have to take place as well. For example, few Japanese fonts are internally encoded according to Han Unification, the principle behind ISO 10646/Unicode. Rather, the internal encoding would be by Kuten numbers or by one of the JIS encodings. If that is also the case for the input encoding, then a double conversion, not always simple, nor necessary, would have to take place.

To make matters even worse, one's editor may not always have the right fonts for a particular language. Transliteration becomes a necessity. But transliteration is completely independent from character encodings; after all, the same Latin transliteration for Cyrillic can be used if one is using ISO 646 or ISO 10646/Unicode. Nor does transliteration have anything to do with font encodings. After all, one would want to use the same Arabic fonts, whether one is typing using a Latin transliteration in ISO 8859-1, or straight Arabic in ISO 8859-6 or ISO 10646/Unicode.

And, to finish us off, the order of characters in a stream of input may not correspond to the order in which characters are to be put on paper or a screen. For example, as explained in Haralambous 1993a, many Khmer vowels are split in two: one part is placed to the left of a consonental cluster, and the other part is placed to the right. The first author has faced similar problems with Sinhalese (Haralambous 1994).

Finally, we should remember that error and log messages must also be generated, and these may not necessarily be in the same character set as either the input encoding or the internal encoding.

Transliteration and contextual analysis. It seems clear that the only viable internal encoding is the font encoding. However, there is no reason that the conversion from input encoding to internal encoding take but one step. Clearly one can always do this, and in fact, if our fonts are sufficiently large, we can always do all analysis at the ligature level in the font. However, making such a decision prevents us from separating distinct tasks, such as — say, for Arabic — first converting all text to ISO 10646/Unicode, then transliterating, then computing the appropriate form of each letter, and only then having the font's ligature mechanism take over.

In fact, what we propose is to allow any number of filters to be written, and that the output from one filter can become the input to another filter, much like UNIX pipes.

Ω Translation Processes In **Ω**, these filters are called Translation Processes (ΩTPs). Each **Ω**TP is defined by the user in an `.otp` file: with a syntax reminiscent of the `flex` lexical analyzer generator, users can define finite state Mealy automata to transform character streams into other character streams.

These user-defined translations tables are not directly read by **Ω**. Rather, compact representations (`.ctp` files) are generated by the `otptoctp` program. A `.ctp` file is read using the **Ω** primitive `\otp` (see below).

Here is the syntax for a translation file:

`in:`	n;
`out:`	n;
`tables:`	T^*
`states:`	S^*
`aliases:`	A^*
`expressions:`	E^*

where n means any number. Numbers can be either decimal numbers, WEB octal (`@'`...) or hexadecimal (`@"`...) numbers, or visible ISO 646 characters enclosed between a GRAVE ACCENT and an APOSTROPHE (`` ` ``*c*`'`).

The first (second) number specifies the number of octets in an input (output) character (the default for both is 1). These numbers are necessary to specify the translation processes that must take place when converting to or from character sets that use more than one octet per character.

Tables are regularly used in character set conversions, when algorithmic approaches cannot be simply expressed. The syntax for a table T is:
id`[`n`] = {`$n, n, \ldots, n$`};`

The **Ω**TPs, as in `flex`, allow a number of states. Each expression is only valid in a given state. The user can specify when to change states. States are often used for contextual analysis. The syntax for a set S of states is:
id`,` *id*`,` `...,` *id*`;`

Expressions are pattern-action pairs. Patterns are written as simple regular expressions, which can be aliased. The syntax for an alias A is:
id `=` L`;`
where L is a pattern.

If only one state is used, then an expression E consists of a pattern and an action:
L `=>` R^*`;`
where the syntax for patterns is:

L	::=	n	
	|	n–n	range
	|	`.`	wildcard
	|	LL	concatenation
	|	$L\{n, m\}$	occurrences
	|	`(`L `|` ... `|` L`)`	choice
	|	`^ (`L `|` ... `|` L`)`	negative choice
	|	`{`*id*`}`	abbreviation;

and where the simplified syntax for actions is:

R	::=	*string*	
	|	n	
	|	$\backslash n$	
	|	$\backslash(\$ - n)$	
	|	$\backslash(* + n - n)$	
	|	`#(`R`)`	
	|	*id*`[`R`]`	
	|	R *op* R	arithmetic;

Patterns are applied to the input stream. When a pattern has matched the input stream, the action to the right is executed. A *string* is simply put on the output stream. The $\backslash n$ refers to the n-th matched character and the $\backslash\$$ refers to the last matched character. The $\backslash *$ refers to the complete matched substream, while $\backslash(* - n)$ refers to all but the last n characters. Table lookup is done using square brackets. All computations must be preceded by a `#`.

Here is an example translation from the Chinese GB 2312-80 encoding to ISO 10646/Unicode:

```
in:  1;
out: 2;
tables: tabgb[8795] = {...};
expressions:
(@"00-@"A0)               => \1;
(@"A1-@"FF)(@"A1-@"FF)  =>
     #(tabgb[(\1-@"A0)*@"64 + (\2-@"A0)]);
. .                        => @"FFFD;
```

where we use `0xfffd` (REPLACEMENT CHARACTER) as the error character.

And here is a common transliteration in Indic scripts:

```
{consonant}{1,6} {vowel}  => \$ \(*-1);
```

The vowel at the end is placed before the stream of consonants.

The complete syntax for expressions is more complicated, as there can be several processing states. In addition, it is possible to push back values onto the input stack. Here is the complete syntax:

<*state*> L `=>` R^* `<=` R^* <*newstate*>

The *state* means that if the ΩTP is in that state then this pattern-action pair can possibly be used. The *newstate* designates the new state if this pattern-action pair is chosen.

Here is an example from the contextual analysis of Arabic:

```
<MEDIAL>{QUADRIFORM}{NOT_ARABIC_OR_UNI}
        => #(\1 + @"DD00)
        <= \2
        <pop:>
        ;
```

When in state `MEDIAL` (in the middle of a word) and a letter that comes in four possible forms is followed by a non-Arabic letter, then the output is the quadriform letter plus the value `@"DD00`. The non-Arabic letter is placed back on the input stack. Then the current state is popped and the ΩTP returns to the previous state, whatever it was.

Loading ΩTPs. Loading an ΩTP is similar to loading a font. The instruction is simply:

`\otp`*newname* = *filename*

The `.ctp` file ***filename***`.ctp` is read in and stored in the otp info memory, similar to the font info memory. A number is assigned to the control sequence *newname*, as for fonts. Thereafter, one can refer to that ΩTP either through the generated number or through the newly defined control sequence.

Input encodings. When reading a file from an unknown source, using an unknown character set, some sort of mechanism is necessary to determine what the character set is. There are two possibilities. Either use a default character set or have some way of quickly recognizing what the character set is.

Fortunately, most character sets contain ISO 646 as a subset. The ISO 10646/Unicode character set, both in its 16-bit and 32-bit versions, retains ISO 646 as its original 128 characters. The only widely used character set that does not fit this mold is IBM's EBCDIC.

We therefore provide the means for automatically detecting the character set family. It suffices that the user place a comment at the very beginning of each file: the `%` character is sufficient to distinguish each of the families. A file using an eight-bit extension of ISO 646 begins with the character code `0x25`; a file with 16-bit characters begins with `0x00 0x25`[1]. Finally, a file using the EBCDIC encoding begins with `0x6c`.

Should there be no comment character, then the default input encoding (ISO 646) is assumed.

Once Ω knows how to read the basic Latin letters, it is possible to *declare* what translation the input must undergo. This is done using the

`\InputTranslation`

command. For example,

`\InputTranslation 1`

states that the entire input stream, starting immediately *after* the newline at the end of this line, will pass through the first ΩTP process.

It is also possible to change the character set within a file. This process is more difficult, as it is not always clear where *exactly* the change is to take place. Suppose that we pass from an 8-bit character set to a 16-bit character set. It is important that we know what is the *last* eight-bit character and what is the *first* sixteen-bit character.

This question can be resolved by specifying a particular character as being the one which changes. However, to simplify matters, we assume that all input translation changes take place *immediately after* the newline at the end of the line in which the `\InputTranslation` appears.

Transliteration. Once characters have been read, most likely to some universal character set such as ISO 10646/Unicode, then contextual analysis can take place, independently of the original character set. This analysis might require several filters, each of which is similar to the translation process undergone by the input.

Since the number of filters that we might want to use is arbitrarily large, there are two commands to specify filters:

`\NumberInputFilters` n

states that the first n input filters are active. The output from the i-th filter becomes the input for the $i+1$-th filter, for $i < n$.

`\InputFilter` m i

states that the m-th input filter is the i-th ΩTP.

Sequences of characters with character codes 5, 10, 11 and 12 successively pass through the translation processes n translation processes. It should be understood that the result of the last translation process should be the font encoding itself; it is in this encoding that the hyphenation algorithm is applied.

Our Arabic example looks like this:

[1] A file with 32-bit characters would begin with `0x00 0x00 0x00 0x25`, but the current version of Ω does not support 32-bit characters.

```
\otp\trans      = ISO646toISO10646
\otp\translit  = TeXArabicToUnicode
\otp\fourform  = UnicodeToContUnicode
\otp\genoutput = ContUnicodeToTeXArabicOut
\InputFilter 0 \translit
\InputFilter 1 \fourform
\InputFilter 2 \genoutput
\NumberInputFilters 3
```

The `TeXArabicToUnicode` translator takes the Latin transliteration and converts it into Arabic. As for `UnicodeToContUnicode`, it does the contextual analysis for Arabic, i.e., it takes Arabic (in ISO 10646/Unicode) and, using a private area, determines which of the four forms (isolated, initial, medial or final) each consonant should take. Finally, `ContUnicodeToTeXArabicOut` determines what slot in the font corresponds to each character. Of course, nothing prevents the font from having its own sophisticated ligature mechanism as well.

Output and special encodings. TeX does not just generate `.dvi` files. It also generates `.aux`, `.log` and many other files, which may in turn be read by TeX again. It is important that the output mechanism be as general as the input mechanism. For this, we introduce the dual operations:

```
\OutputTranslation
\OutputFilter
\NumberOutputFilters
```

with, of course, the appropriate arguments.

Similarly, TeX can output, in its `.dvi` files, commands that are device-driver specific, using `\special` commands. Since the arguments to `\special` are themselves strings, it seems appropriate to also allow the following commands:

```
\SpecialTranslation
\SpecialFilter
\NumberSpecialFilters
```

Large fonts TeX is limited to fonts that have a maximum of 256 characters. However, on numerous occasions, a need has been shown for larger fonts. Obviously for languages using ideograms, 256 characters is clearly not sufficient. However, the same holds true for alphabetic scripts such as Latin, Greek and Cyrillic; for each of these, ISO 10646/ Unicode defines more than 256 pre-composed characters. However, many of these characters are basic character-diacritic mark combinations, and so the actual number of basic glyphs is quite reduced. In fact, for each of these three alphabets, a single 256-character font will suffice for the basic glyphs.

We have therefore decided, as a first step, to offer the means for large (16-bit) virtual fonts, whose basic glyphs will reside in 8-bit real fonts. This is clearly only a first step, but it has the advantage of allowing large fonts, complete with ligature mechanisms, without insisting that all device drivers be rewritten.

In addition to changing TeX, we must also change `dvicopy` and `vptovf`, which respectively become `xdvicopy` and `xvptoxvf`. The `.tfm`, `.vp` and `.vf` files are replaced by `.xfm`, `.xvp` and `.xvf` files, respectively. Of course, the new programs can continue to read the old files.

`.xfm` files. The `.xfm` files are similar to `.tfm` files, except that most quantities use 16 or 32 bits. Essentially, most quantities have doubled in size. The header consists of 13 four-octet words. To distinguish `.tfm` and `.xfm` files, the first word is all zero. The next eleven words are the values for *lf*, *lh*, *bc*, *ec*, *nw*, *nh*, *nd*, *ni*, *nl*, *nk*, *ne*, and *np*. All of these values must be non-negative and less than 2^{31}. Now, each *char_info* value is defined as follows:

width index	16 bits
height index	8 bits
depth index	8 bits
italic index	14 bits
tag	2 bits
remainder	16 bits

Each *lig_kern_command* is of the form:

op byte	16 bits
skip byte	16 bits
next char	16 bits
remainder	16 bits

Finally, extensible recipes take double the room.

`.xvp` files. The `.xvp` files are simply `.vpl` files in which all restrictions to 8-bit characters have been removed. Otherwise, everything else is identical.

Minor changes. Since the changes above required carefully passing through all of the code for TeX, we took advantage of the opportunity to remove all restrictions to a single octet. So, for example, more than 256 registers (of each kind) can be used. Similarly, more than 256 fonts can be active simultaneously.

Applications of Ω

Dutch, Portuguese, Turkish: the easy way. These three languages (and maybe others?) have at least one thing in common: they need fonts with a slightly different ligature table than the one in the Cork encoding. Dutch typesetting uses the notorious 'ij' ligature (think of the names of people very well known to us: Dijkstra, Eijkhout, van Dijk, or the name of the town Nijmegen or the lake IJselmeer); this ligature appears in the Cork encoding (as well as in ISO 10646/Unicode), but until now there was no user-transparent means of obtaining it. In **Ω** you just need to place a macro calling a specific ΩTP filter into the expansion of the Dutch-switching macro; according to **Ω** syntax (described in section) this ΩTP can be written as simply as

```
in: 1
out: 2
expressions:
`I'`J' => @"0132;
`i'`j' => @"0133;
.       => \1;
```

where `0x0132` and `0x0133` are characters 'IJ' (LATIN CAPITAL LIGATURE IJ) and 'ij' (LATIN SMALL LIGATURE IJ) in ISO 10646/Unicode.

Portuguese and Turkish typesetting do not use ligatures 'fi',... 'ffl' (in Turkish there is an obvious reason for doing this: the Turkish alphabet uses both letters 'i' and 'ı', and hence it would be impossible to know if 'fi' stands for 'f' + 'i' or 'f' + 'ı'). This is a major problem for TEX, since the only solution that would retain a natural input would be to use new fonts; and defining a complete new set of fonts (either virtual or real), just to avoid 5 ligatures, is more trouble than benefit. **Ω** solves that problem easily; of course, it is not possible to 'disable' a ligature, since the latter arrives at the very last step, namely inside the font. It follows that we must cheat in some way; the natural way is to place an invisible character between 'f' and 'i'; in ISO 10646/Unicode there is precisely such a character, namely `0x200b` (ZERO WIDTH SPACE); this operation could be done by an ΩTP line of the type `` `f'`i' => "f" @"200b "i" `` for each ligature. This character would then be sent to the Cork table's 'compound mark' character, which was defined for that very reason.

A still better way to do this would be to define a second 'f' in the output font table, which would *not* form a ligature with 'f', 'i', or 'l'. This would give the font the possibility of applying a kern between the two letters, and counterbalance the effect of the missing ligature (after all, if a font is designed to use a ligature between 'f' and 'i', a non-ligatured 'fi' pair would look rather strange and could need some correction).

Adobe Poetica. Poetica is a chancery script font, designed by Robert Slimbach and released by Adobe Systems Inc. Stating Adobe's promotional material, *"The Poetica typeface is modeled on chancery handwriting scripts developed during the Italian Renaissance. Elegant and straightforward, chancery writing is recognizable as the basis for italic typefaces and as the foundation of modern calligraphy. Robert Slimbach has captured the vitality and grace of this writing style in Poetica. Characteristic of the chancery hand is the common use of flourished letterforms, ligatures and variant characters to embellish an otherwise formal script. To capture the variety of form and richness of this hand, Slimbach has created alternate alphabets and character sets in his virtuoso Poetica design, which includes a diverse collection of these letterforms."*

Technically, the Poetica package consists of 21 PostScript fonts: Chancery I–IV, Expert, Small Caps, Small Caps Alternate, Lowercase Alternates I–II, Lowercase Beginnings I–II, Lowercase Endings I–II, Ligatures, Swash Caps I–IV, Initial Swash Caps, Ampersands, Ornaments. The ones of particular interest for us are Alternate, Beginnings, Endings and Ligatures, since characters from these can be chosen automatically by **Ω**. The user just types plain text, possibly using a symbol to indicate degrees of alternation. An ΩTP converts the input into characters of a (virtual) 16-bit font, including the characters of all Poetica components. Using several ΩTPs and changing them on the fly will allow the user to choose the number of ligatures he/she will obtain in the output. It will also allow us to go farther than Adobe, and define kerning pairs between characters from different Poetica components.

See Fig. 1, for a sample of text, typeset in Poetica.

On Greek, ancient and modern (but rather ancient than modern).

Diacritics against kerning. It is in general expected of educated men and women to know Greek letters. Already in college, having used θ for angles, γ for acceleration, and π to calculate the area of a round apple pie of given radius, we are all familiar with these letters, just as with the Latin alphabet. But the Greek language, in particular the ancient one, needs more than just letters to be written. Two kinds of diacritics are used, namely accents (acute, grave and circumflex), and breathings (smooth and raw) which are placed on vowels; breathings are also placed on the consonant rho.

Every word has at most one accent[2], and 99.9% of Greek words have exactly *one* accent. Every word starting with a vowel has exactly one breathing[3]. It

[2] Sometimes an accent is transported from a word to the preceding one: ἄνθρωπός τις instead of ἄνθρωπος, τίς, so that typographically a word has more than one accents.

[3] With one exception: the letters ρρ are often written ῤῥ, when inside a word: ποῤῥῶ.

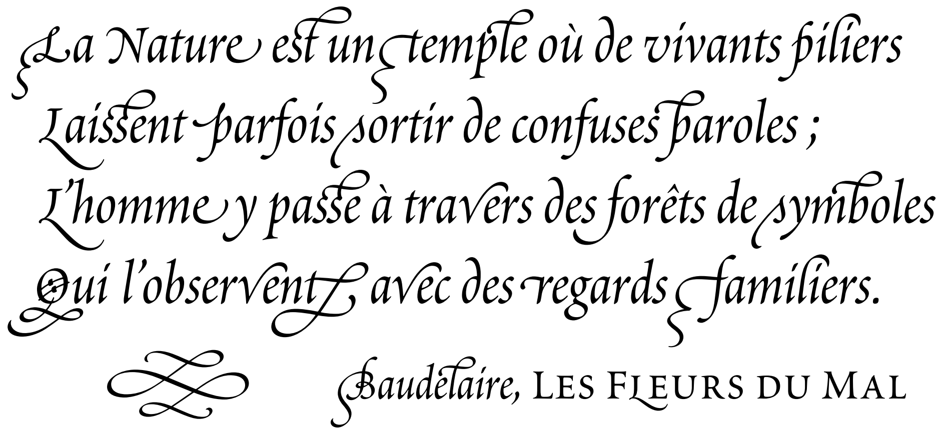


Figure 1: Sample of the Poetica typeface.

follows that writing in Greek involves much more accentuation than any Latin-alphabet language, with the obvious exception of Vietnamese.

How does TEX deal with Greek diacritics? If the traditional approach of the `\accent` primitive had been taken, then we would have practically *no* hyphenation (which in turn would result into disastrous over/underfulls, since Greek can easily have long words like ὠτορινολαρυγγολογικός), *no* kerning, and a cumbersome input, involving one or two macros for every word.

The first approach, originated by Silvio Levy (Levy 1988), was to use TEX's ligatures ('dumb' ones first, 'smart' ones later on) to obtain accented letters out of combinations of codes representing breathings (`>` and `<`), accents (`‘`, `’` and `~` or `=`) and the letters themselves. In this way, one writes `>‘h` to get ἢ. This approach solved the problem of hyphenation and of cumbersome input.

Nevertheless, this approach fails to solve the *kerning* problem. Let's take the very common case of the article τὸ (letter tau followed by the letter omicron); in almost all fonts there is a kerning instruction between these letters, obviously because of invariant characteristics of their shapes. Suppose now that omicron is accented, and that one writes `t‘o` to get tau followed by omicron with grave accent. What TEX sees is a 't' followed by a grave accent. No kerning can be defined for these two characters, because we have no idea what may follow after the grave accent (it can very well be a iota, and usually there is no kerning between tau and iota). When the letter omicron arrives, it is too late; TEX has already forgotten that there was a tau before the grave accent.

A solution to this problem would be to write diacritics *after* vowels ("post-positive notation"). But this contradicts the visual characteristics of diacritics in the upper case, since these are placed on the left of uppercase letters: Ἔαρ could hardly be transliterated `E>’ar`. And, after all, TEX should be able to do proper Greek typesetting, however the letters and diacritics are input.

Ω solves this problem by using an appropriate chain of Ω Translation Processes (ΩTPs), a notion explained in section : as example, consider the word ἔαρ:

1. Suppose the user wishes to input his/her text in 7-bit ASCII; he/she will type `>’ear`, and this is already ISO-646, so that no particular input translation is needed. Another choice would be to use some Greek input encoding, such as ISO-8859-7 or ΕΛΟΤ; then he/she may as well type >᾿εαρ or >έαρ.[4] The first ΩTP will send these codes to the appropriate 16-bit codes in ISO 10646/Unicode: `0x1f14` for ἔ (GREEK SMALL LETTER EPSILON WITH PSILI AND OXIA), `0x03b1` for α (GREEK SMALL LETTER ALPHA), and `0x03c1` for ρ (GREEK SMALL LETTER RHO).
2. Once Ω knows what characters it is dealing with (Ω's default internal encoding is precisely ISO 10646/Unicode), it will hyphenate using 16-bit patterns.
3. Finally, an appropriate ΩTP will send Greek ISO 10646/Unicode codes to a 16-bit virtual font (see next section to find out why we need 16

[4] The reason for the absurd complication of being forced to type either >᾿ε or >έ to obtain ἔ, is that "modern Greek" encodings have taken the easy way out and feature only one accent, as if the Greek language was born in 1982, year of the hasted and politically motivated spelling reform.

bits), built up from one or more 8-bit fonts. This font contains kerning instructions, applied in a straightforward manner, since we are now dealing with only three codes: <ἔ>, <α> and <ρ>. No auxiliary codes interfere anymore.

4. `xdvicopy`[5] will de-virtualize the `dvi` file and return a new `dvi` file using only 8-bit fonts, compatible with every decent `dvi` driver.

By separating tasks, hyphenation becomes more natural (for TeX one has to use patterns including auxiliary codes `'`, `` ` ``, `=` etc.). Furthermore, an additional problem has been solved *en passant*: the primitives `\lefthyphenmin` and `\righthyphenmin` apply to characters of `\catcode` 12. To obtain hyphenation between clusters involving auxiliary codes, we have to declare these codes as 'letter-like characters'. For example, the word ἔαρ, written as `>'ear`: the codes `>` and `'` must be considered as letter-like (non-trivial `\lccode`) to allow hyphenation; but this means that for TeX, ἔαρ has *5 letters* instead of 3, and hence even if we ask `\lefthyphenmin=3`, we will still get the word hyphenated as ἔ-αρ!! **Ω** solves this problem by hyphenating *after* the translation has been done (in this case `>'e` or >᾿ε or >έ → ἔ).

Dactyls, spondees and 16-bit fonts. Scholarly editions of Greek texts are slightly more complicated than plain ones[6], one of the add-ons being a *third* level of diacritization: syllable lengths.

One reads in Betts and Henry 1989, p. 254: *"Greek poetry was composed on an entirely different principle from that employed in English. It was not constructed by arranging stressed syllables in patterns, nor with a system of rhymes. Greek poets employed a number of different metres, all of which consist of a certain fixed arrangement of* ***long and short syllables****"*. Long and short syllables are denoted by the diacritics *macron* and *breve*. The diacritics are placed *between* the letter and the regular diacritic, if any (except in the case of uppercase letters, where they are placed after over the letter while regular diacritics are placed to its left).[7]

The famous first two verses of the Odyssey

Ἄνδρα μοι ἔννεπε, Μοῦσα πολύτροπον, ὃς μάλα πολλὰ
πλάγχθη ἐπεὶ Τροίης ἱερὸν πτολίεθρον ἔπερσε·

form *hexameters.* These consists of six feets: four dactyls or spondees, a dactyl and a spondee or trochee (see again Betts and Henry 1989 for more details). One could write the text without accents or breathings to make the metre more apparent:

Ᾱνδρᾰ μοῐ ε̄ννε̆πε̆, Μοῡσᾰ πο̆λῡτρο̆πο̆ν, ο̄ς μᾰλᾰ πο̄λλᾰ
πλᾱγχθη̆ ε̆πεῑ Τροῑη̄ς ῐε̆ρο̄ν πτο̆λῐε̄θρο̆ν ε̆πε̄ρσε̆

or one could decide to typeset all types of diacritics:

᾿Ᾱνδρᾰ μοῐ ἔ̄ννε̆πε̆, Μοῦ̄σᾰ πο̆λῡ́τρο̆πο̆ν, ὃ̄ς μᾰ́λᾰ πο̄λλᾰ̀
πλᾰ́γχθη̆ ἐ̆πεὶ̄ Τροῑ́η̄ς ἱ̆ε̆ρο̄̀ν πτο̆λῐ́ε̄θρο̆ν ἔ̆πε̄ρσε̆

Having paid a fortune to acquire the machine that sits between the keyboard and the screen, one could expect that hyphenation of text and kerning between letters remains the same, despite the constantly growing number of diacritics. Actually this isn't possible for TeX: there are exactly 345 combinations of Greek letters with accents, breathings, syllable lengths and subscript iota; TeX can handle at most 256 characters in a font. Therefore **Ω** is necessary for hyphenation of Greek text, whenever syllable lengths are typeset.

In this case, things do not run as smoothly as in the previous section: although 345 is a small number compared to 65,536 (= 2^{16}), ISO-people decided that there is not room enough for all combinations of accents, breathings and syllable lengths.[8]

Whenever ISO 10646/Unicode comes too short for our needs, we use the *private zone.* Just as in the TV serial *The Twilight Zone*, in the private zone *anything* can happen. In the case of **Ω** all operations remain internal, so that we have absolute freedom of defining characters: in ISO 10646/Unicode, the private zone consists of characters `0xe000` to `0xfffd` (of group 0, that is the 16-bit part of ISO 10646/ Unicode), a total of 8,190 positions.

Ω will treat the input like in the previous section, but letters with macron and breve diacritics will occupy internal positions in the private zone. The rest of the treatment will be exactly the same. As for the transliterated input one could take `^` and `_` to denote macron and breve (after having changed their catcodes so that they do not interfere with math operators), or any other combination of 7-bit or 8-bit codes.

[5] In the name of this program, which is an extended version of Peter Breitenlohner's `dvicopy`, '`x`' stands for 'extended', not for 'X-Window'.

[6] After all, scholars have been studying Greek text for more than 2,000 years now.

[7] These additional diacritics are also used for a different purpose: in prose, placed on letters alpha, iota or upsilon, they indicate if they are pronounced long or short (this time we are talking about *letters* and not about *syllables*).

[8] Nevertheless, they included lower and upper alpha, iota and upsilon with macron and breve, probably for the reasons explained in the previous footnote. However, combining diacritics must be used to code letters with macron/breve *and* additional diacritics.

A dream that *may* come true. As stated by the first author in his Cork talk, back in 1990, his dream was—and still is—to draw a Greek font based on the famous « Grecs du Roi » typeface by Claude Garamont, graved in 1544–46 for the king François I. This typeface was designed after a manuscript of Ἄγγελος Βεργήκιος, a Cretan, calligrapher and reader of Greek at the French Court, in the beginning of the XVI century. There are 1327 different types, most of them ligatures of two or more letters (sometimes entire words). One can read in Nationale 1990 that *"this typeface is the most precious piece of the collection [of the French National Printing House]"*, certainly not the least of honors! Ω is the ideal platform for typesetting with this font, since it would need only an additional ΩTP to convert plain Greek input into ligatures. The author hopes to have finished (or, on a more realistic basis, to have brought to a decent level) this project in time for the International Symposium "Greek Letters, From Linear B to Gutenberg and from G to Ψ", which will take place in Athens in late Spring 1995, organized by the Greek Font Society[9].

Arabic, or "The Art of separating tasks".

Plain Arabic, quick and clean and elegant. Arabic typesetting is a beautiful compromise between Western typesetting techniques (finite number of types, repeated *ad infinitum*) and Arabic calligraphy (infinite number of arbitrarily complex ligatures). We can subdivide Arabic ligatures into two categories: (a) mandatory ones: letter connections (ه + م → هم) and the special ligature lām-alīf (ل + ا → لا), and (b) optional ones, used for esthetic reasons.

The second category of ligatures corresponds to our good old 'fi' ,'fl' ,etc. They depend on the font design and on the degree of artistic quality of a document. The author has made a thorough classification of esthetic ligatures of the Egyptian typecase, in Haralambous 1992 (reprinted in (Haralambous 1993c)). Here is an example of the ligaturing process of the word تحمل, following Egyptian typographical traditions:

- ت ح م ل (letters not connected);
- تحمل (only mandatory ligatures, connecting letters);
- تحمل (esthetic ligature between the first two letters);
- تحمل to (esthetic ligature between the first three letters).

To produce more than 1,500 possible ligatures of two, three or four letters, three 256-character tables were necessary. Each ligature is constructed by superposition of small pieces. Once TEX knows which characters to take, and from which font, it only needs to superpose them (no moving around is necessary). The problem is to recognize the existence of a ligature and to find out which characters are needed. This process is highly font-dependent. A different font—for example in Kuffic or Nastaliq style—may have an entirely different set of ligatures, or none at all (like the plain font, in which the two words تيخ العربى are written, that is widely used in computer typesetting because of its readability); nevertheless, the mandatory ligatures remain strictly the same, whatever font is used.

Up to now, there are three solutions to the problem of mandatory Arabic ligatures:

- First, by K. Lagally Lagally 1992, is to use TEX macros for detecting and applying mandatory ligatures (we say: "to do the contextual analysis"). This process is cumbersome and long. It is highly dependent on the font encoding and the macros used can interfere with other TEX macro packages. All in all, it is not the natural way to treat a phenomenon which is a low-level fundamental characteristic of the Arabic script.
- Second, by the first author Haralambous 1993b, is to use "smart" TEX font ligatures (together with TEX--XET, the bi-directional version of TEX); this process is philosophically more natural, since contextual analysis is done behind the scenes, on the very lowest level, namely the one of the font itself. It does not depend on the font encoding, since every font may use its own set of ligatures. The disadvantage lies in the number of ligatures needed to accomplish the task: about 7,000! The situation becomes tragic when one wants to use a dozen Arabic fonts on the same page: TEX will load 7,000 probably strictly identical ligatures for each font. You need more than BigTEX to do that.
- Third, also by the first author Haralambous 1993b, is to use a preprocessor. The advantage is that the contextual analysis is done by a utility dedicated to this task, with all possible bells and whistles (for example adding variable length connecting strokes, also known as

[9] For additional information on the Symposium, contact Ἑταιρεῖα Ἑλληνικῶν Τυπογραφικῶν Στοιχείων, Ἑλλανίκου 39-41, 116 35 Ἀθήνα, Greece, or Michael S. Macrakis, 24 Fieldmont Road, Belmont, MA 02178-2607, USA.

'keshideh'); it is quick and uses only a very small amount of memory. Unfortunately there are the classical disadvantages of having a preprocessor treat a document before TEX: one file may `\input` another file, from any location of your net, and there is no way to know in advance which files will be read, and hence have to be preprocessed; preprocessor directives can interfere with TEX macros; there is no nesting between them and this can easily produce errors with respect to TEX grouping operations, etc.

None of these methods can be applied for large scale real-life Arabic production: in all cases, the Arabic script is treated as a 'puzzle to solve' and, inevitably, TEX's performance suffers.

We use ΩTPs to give a natural solution to the problem; consider once again the example of the word تحمل:

- First, تحمل is read by Ω, either in Latin transliteration (`tHml`) or in ISO 8859-6, or ASMO, or Macintosh Arabic, or any decent Arabic script input encoding.
- The first ΩTP converts this input into ISO 10646/Unicode codes for Arabic letters: `0x062a` (ت, ARABIC LETTER TEH), `0x062d` (ح, ARABIC LETTER HAH), `0x0645` (م, ARABIC LETTER MEEM), `0x0644` (ل, ARABIC LETTER LAM);
- ISO 10646/Unicode being a *logical* way of codifying Arabic letters, and not a graphical one, there is no information on their contextual forms (isolated, initial, medial, final). The second ΩTP sends these codes to the private zone, where we have (internally) reserved positions for the combinations of Arabic characters and contextual forms. Once this is done, Ω knows the form of each character.
- The third ΩTP simply translates these codes to a 16-bit standard Arabic TEX font encoding (this is a minor operation: the private zone being located at the end of the 16-bit table, we move the whole block near to the beginning of the table).
- If the font has no esthetic ligatures, we are done: Ω will send the results of the last ΩTP to the `dvi` file, and produce تحمل. On the other hand, if there are still esthetic ligatures—as in تحمل —then these will be included in the font, as 'smart' ligatures. Since the font table can have as many as 65,536 characters, there is plenty of room for small character parts to be combined.[10]

What we have achieved is that the fundamental process of contextual analysis is done by background machinery (just like TEX hyphenates and breaks paragraphs into lines), and that the optional esthetic refinements are handled exclusively by the font (in analogy to Roman fonts having more ligatures than typewriter ones, etc.).

Vowelized Arabic (things get harder). In plain contemporary Arabic, only consonants and long vowels are written; short vowels have to be guessed by the reader, using the context (the same consonants with different short vowels, can be understood as a verb, a noun, an adjective etc.). When it is essential to specify short vowels, small diacritics are added over or under the letters. Besides short vowels, there are also diacritics for doubling consonants, for indicating the absence of vowel, and for the glottal stop (like in "Oh-oh"): counting all possible combinations, we obtain 14 signs. These diacritics can give TEX a hard time, since they have to be coded between consonants, and hence interfere in the contextual analysis algorithm: for example, suppose that TEX is about to typeset letter x, which is the last letter of a word, followed by a period. Having read the period, TEX knows that the letter has to be of final form (one of the 7,000 ligatures has to be $<x$ in medial form$> + <.> \rightarrow <x$ in final form$><.>$). Now suppose that the letter is immediately followed by a short vowel, which in our case is necessarily placed between the letter x and the period. TEX's smart ligatures cannot go two positions backwards; when TEX discovers the period after the short vowel it is too late to convert the medial x into a final one.

Fortunately, ΩTPs are clever enough to be able to calculate letter forms, whatever the diacritics surrounding them (which is exactly the attitude of the human typesetter, who first typesets the letters, and then adds the corresponding diacritics).

Nevertheless, ΩTPs are not perfect, and there are problems which cannot be solved, even by the most judicious ΩTP: for example, the positioning of diacritics. We all know that TEX (and hence also Ω, which is no more than an humble extension of TEX) places all elements on a page by using boxes. Unfortunately, the placement of diacritics requires

[10] If these esthetic ligatures are used in several fonts, it might be possible that we have the same problem of overloading Ω's font memory; in this case, we can always write a fourth ΩTP, which would systematically make the 'esthetic analysis' out of the contextually analyzed codes.

more information than just the height, width, depth and italic correction of a character; in some cases, a real insight into the shape of the character itself and the surrounding characters is necessary (think of ligatures constructed vertically out of four letters, each one having its own diacritic).

This problem can easily be solved for an (esthetically) non-ligatured font: counting all possible letters (no to forget Farsi, Urdu, Pashto, Sindhi, Kirghiz, Uigur and other languages using variants of Arabic letters), in all possible forms one may end up with a figure of no more than a thousand glyphs. Combining these glyphs with the 14 diacritical signs, combinations would result in no more than 14,000 font positions, a figure well under the 65,536 character limit. Since the private ISO 10646/Unicode zone is not big enough to handle so many characters, we would use an additional ΩTP to send combinations of <contextually analyzed consonant or long vowel> and <diacritic> to codes in the output font encoding. The advantage of this method is that every diacritic can be placed individually (assuming a minute to find the ideal position for a diacritic is needed, the font can be completed in four weeks of steady work), or one can use QDTEXVPL methods to place the diacritics automatically, and then make the necessary corrections.

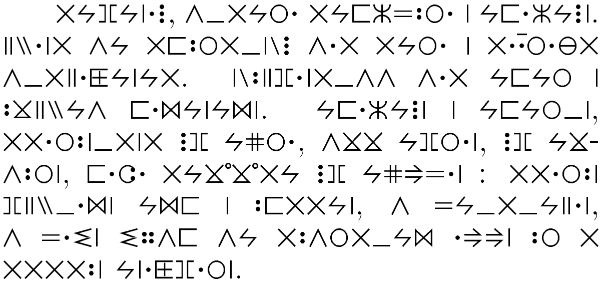
Unfortunately the number of necessary font positions grows astronomically when we consider 2- or 3-letter ligatures, where each letter can have a different diacritic. One of the future challenges of the Ω project will be to analyze Arabic script characters and find the necessary parameters to determine diacritic positioning, just like D.E. Knuth did for mathematical typesetting. It should be noted that despite the huge complexity of this task, we remain in the strict domain of Arabic *typography*, which is after all no more than a *reduced and simplified version of Arabic calligraphy.*

Multiscript languages ("can you *read* Vietnamese?"). Both Westerners and Arabs had the—not so democratic—habit of imposing their alphabet on nations they conquered (either militarily, religiously, culturally or technologically). So it happens that we can all read Vietnamese (but not understand it) just as any Arab can read Malay and Sindhi, and not understand a word (except perhaps for some Arabic words which accompanied the alphabet in its journey).

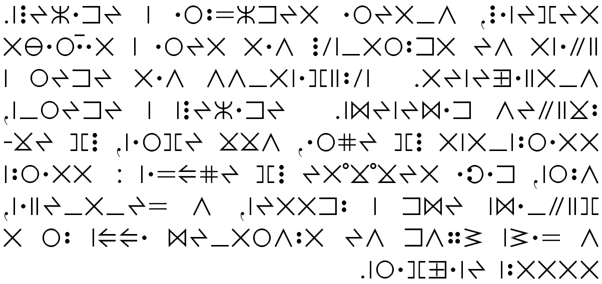
In some cases, more than one script remained in use for the same language and attempts are made to clarify and standardize the equivalences between letters of these scripts, in order to provide an efficient transliteration algorithm.

The first author has worked on two cases involving the Arabic script: Berber and Comorian.

Berber: a language with three scripts and two writing directions. The Berber language (Berbers call it "Tamazight") can be written in Arabic, Latin or native ("Tifinagh") script. The first author has developed, under the guidance of Salem Chaker,[11] a Tifinagh font in METAFONT. Here is a small sample of Berber text, written in left-to-right Tifinagh:



Follows the same text in right-to-left Tifinagh:



and in Arabic script:

تيفينــاغ، دتيرا تيمــزوّرا ن يِمازيغـ$. لاّ#ت
دي تمورتنّغ دات تيــــــــــــرا ن تاعرابت
دتلاطينيـــت. نّولفا#ـــتدّ دات يِميِر ن وقلّيد
ماسينيسـ$. يِمازيغ$ ن يِميــرن، تّارونتنــت
غف يِزرا، دقّ يِفران، غف يِقدورن، ماشــــــا
تيقُّتي غـف يِزكوان : تّارون فلّاسـ$ يِسـم ن
ومتّيـــــــــــ$، د ويتيلان، د وايـ$ يخدم دي
تودرتيس اكّ$ ور ت تّتّون يِناطفارن.

The encoding of these fonts is such that one can output the same TEX input in Latin transliteration, left-to-right Tifinagh, right-to-left Tifinagh or Arabic. All one needs to change is a macro at the beginning. Accomplishing this feature for Latin

[11] Director of the Berber Studies Department of the National Institute of Oriental Languages in Paris.

and Tifinagh was more-or-less straightforward; not so for Arabic. Unfortunately, that font has all the problems of plain Arabic fonts: it needs more than 7,500 ligatures to do the contextual analysis, and is overloaded: there is no longer room to add a single character, an annoyance for a language that is still under the process of standardization.

Another source of difficulties is the fact that the equivalences between Latin, Tifinagh and Arabic are not immediate. Some short vowels are written in the Latin text, but not in the Arabic and Tifinagh ones. Moreover, double consonants are written explicitly in Latin and Tifinagh, while they are written as a single consonant with a special diacritic in Arabic. And perhaps the most difficult problem is to make every Berber writer feel "at home" , regardless of the script he/she uses: one should not have the impression that one script is privileged over the others!

Finally, the last problem (not a minor one when it comes to real-life production), is that we need a special Arabic font for Berber, because of the different input transliteration: for example,while in plain Arabic transliteration we use 'v' for ڤ and 'sh' for ش, in Berber we are forced to use 'g' for the former and 'c' for the latter. There are two supplementary letters used for Berber in Arabic script: ڃ and ږ; these letters are also used in Sindhi and Pashto, so that the glyphs are already covered by the general Arabic TEX system; but in Berber, they have to be transliterated as 'j' and 'z', because of the equivalences with the Latin alphabet. This forces us to use a different transliteration scheme than the one for plain Arabic, and hence—because of TEX's inability to clearly separate input and output encoding—to use a differently encoded TEX output font. Imagine you are typesetting a book in both Berber and Arabic; you will need two graphically identical fonts for every style, point size, weight and font family, each one with more than 7,000 ligatures. And we are just talking of (esthetically) non-ligatured fonts!

Ω solves this problem by using the same output fonts for Berber and plain Arabic. We just need to replace the first **Ω**TP of the translation chain: the one that converts raw input into ISO 10646/Unicode codes. Berber linguists can feel free to invent/introduce new characters or diacritics; as long as they are included in the ISO 10646/Unicode table we will simply have to make a slight change to the first **Ω**TP (and if these signs are not yet in ISO 10646/Unicode we will use the private zone).

Comorian: African Latin versus Arabic. A similar situation has occurred in the small islands of the Comores, between Madagascar and the African continent. Both the Latin alphabet (with a few additions taken from African languages) and the Arabic one are used. Because of the many sounds that must be distinguished, one has to use diacritics together with Arabic letters. These diacritics look like Arabic diacritics (for practical reasons) but are not used in the same way; in fact, they are part of the letters, just like the dots are part of plain Arabic letters.

Once again, the situation can easily be handled by an **Ω**TP. While it is still not clear what should be proposed for insertion into ISO 10646/Unicode (this proposal, made by Ahmed-Chamanga, a member of the Institute of Oriental Languages in Paris, is now circulating from Ministries to Educational and Religious Institutions, and is being tested on natives of all educational levels), Comorians can already use **Ω** for typesetting, and upgrade the transliteration scheme on the fly.

Khmer As pointed out in Haralambous 1993a, the Khmer script uses clusters of consonants, subscript consonants, vowels and diacritics. Inside a cluster, these parts have to be moved around by TEX to be positioned correctly. It follows that TEX *must* use `\kern` instructions between individual parts of a cluster. Because of these, there is no kerning anymore: suppose that characters ខ and ក need to be kerned; and suppose that the consonant ខ is (logically) followed by the subscript consonant ្ក, which is (graphically) placed under this letter: ខ្កក. For TEX, ខ is not immediately followed by ក anymore, and so no kerning between these letters will be applied; nevertheless, graphically they still *are* adjacent, and hence need eventually to be kerned.

Ω uses the sledgehammer solution to solve this problem: we define a 'big' (virtual) Khmer font, containing *all currently known* clusters. As already mentioned in Haralambous 1993a, approximately 4,000 codes would be sufficient for this purpose. And of course one could always use the traditional TEX methods to form exceptional clusters, not contained in that font.

Like in Arabic, we deal with Khmer's complexity by separating tasks. A first **Ω**TP will send the input method the use has chosen, to ISO 10646/Unicode Khmer codes (actually there aren't any ISO 10646/Unicode Khmer codes yet, but the first author has submitted a Khmer encoding proposal to the appropriate ISO committees, and expects that soon some step will be taken in that direction—for the moment we will use once again the private zone). A second **Ω**TP will analyze these codes contextually and will send groups of them to the appropriate cluster codes. The separation of tasks is essential for allowing multiple input methods, without redefining each time the contextual analysis—which is after all a basic characteritics of the Khmer writing system. Ω will send the result of the second ΩTP to the `dvi` file, using kerning information stored in the (virtual) font. Finally, `xdvicopy` will de-virtualize the `dvi` file and create a new file using exclusively characters from the original 8-bit Khmer font, described in Haralambous 1993a.

iso 10646/Unicode and beyond. It is certainly not a trivial task to fit together characters from different scripts and to obtain an optically homogeneous result. Often the esthetics inherent to different writing systems do not allow sufficient manipulation to make them 'look alike'; it is not even trivial if this should be tried in the first place: suppose you take Hebrew and Armenian, and modify the letter shapes until they resemble sufficiently, to our Western eyes. It is not clear if Armenian would still look like Armenian, or Hebrew like Hebrew; furthermore one should not neglect the psychological effects for the reader when switching between scripts (and all other changes the change of scripts implies: language, culture, state of mind, idiosyncracy, background): the more the scripts differ, the easier this transition can be made.

The only safe thing we can do with types from different origins is to balance stroke widths so that the global gray density of the page is homogeneous (no "holes" inside the text, whenever we change scripts).

These remarks concern primarily scripts that have significantly different esthetics. There is one case, though, where one can apply all possible means of uniformization, and letters can be immediately recognized as belonging to the same font family: the LGCAI group (LGCAI stands for "Latin, Greek, West/East Cyrillic, African, Vietnamese and IPA"). Very few types cover the entire group: Computer Modern is one of them (not precisely the most beautiful), Unicode Lucida another (a nice Latin font but rather a misfailure in lowercase Greek); there are Times fonts for all members of this group, but there is no guarantee that they belong to the same Times style, similarly for Helvetica and Courier. Other adaptations have been tried as well and it is to be expected that the success (?) of Windows NT will lead other foundries into "extending" their typefaces to the whole group[12].

Fortunately, TEX/Ω users can already now typeset in the whole LGCAI range, in Computer Modern[13] (by eventually adding a few characters and correcting some others). The 16-bit font tables of Ω allow:

1. hyphenation patterns using arbitrary characters of the group;
2. the possibility of avoiding frequent font changes, for example when switching from Turkish to Welsh, to Vietnamese, to Ukrainian, to Hawsa;
3. potential kerning between all characters.

But Ω can go even beyond that: one can include different styles in the same (virtual) font; looking at the ISO 10646/Unicode table, one sees that LGCAI characters, together with all possible style-dependent dingbats and punctuations, fit in 6 rows (1,536 characters). This means, that—at least theoretically—a virtual Ω font can contain up to 42 (!) style variations[14] of the whole LGCAI group, for example Italic, Bold, Small Caps, Sans Serif, Typewriter and all possible combinations [a total of 24 = (Roman or Italic) × (normal or bold) × (plain or small caps[15]) × (Serif or Sans Serif or Typewriter)]. Defining kerning pairs between all different styles will avoid the use and misuse of \/ (italic correction) and give a better appearance to mixed-style text.

It will be quite an experience to make such a font, since many African and IPA characters have no uppercase or italic or small caps style defined yet; see Jörg's Knappen paper on African fonts (Knappen 1993) for the description of a few problematic examples and the solutions he proposes.

[12] As a native African pointed out to the first author, this will also mean that Africans will find themselves in the sad and paradoxal situation of having (a) fonts for their languages, (b) computers, since Western universities send all their old equipment to the Third World, but (c) no electricity for running them and using the fonts…

[13] *Definitely, sooner or later some institution or company will take the highly praisable initiative of sponsoring the development of other METAFONT typefaces*; the authors would like to encourage this idea.

[14] The authors would like to emphasize the fact that it is by no means necessary to produce all styles out of the same METAFONT code, as is done for Computer Modern. As a matter of fact the font we are talking about can very well be a mixture of Times, Helvetica, Courier; the advantage of having them inside the same structure is that we can define kerning pairs between characters of different styles.

[15] The figure is actually less than 18, since only lowercase small caps are needed…

Other applications: *Ars Longa, Vita Brevis.* In this paper we have chosen only a few applications of **Ω**, out of personal and highly subjective criteria. Almost every script/language can take advantage in one or another way of the possibilities of internal translation processes and 16-bit tables. For example, the first author (cf. Haralambous 1994) has presented at the 1994 TUG Meeting the pre-processor *Indica* for Indic languages (languages of the Indian subcontinent, except Urdu). *Indica* will be rewritten as a set of **Ω**TPs; in this way we will eliminate all problems of preprocessing TEX code.

All in all, the development of 16-bit typesetting will be a fascinating challenge in the next decade, and TEX/Ω can play an important rôle, because of it's academic support, openness, portability, and non-commercial spirit.

References

Beeton, B. "Mathematical symbols and Cyrillic fonts ready for distribution (revised)". *TUGboat* **6**(3), 124–128, 1985.

Betts, G. and Henry, A. *Teach yourself Ancient Greek.* Hodder and Stoughton, Kent, 1989.

Haralambous, Y. "Typesetting the Holy Qūran with TEX". In *Proceedings of the 2nd International Conference on Multilingual Computing—Arabic and Latin script (Durham).* 1992.

Haralambous, Y. "The Khmer Script tamed by the Lion (of TEX)". In *Proceedings of the 14th TEX Users Groups Annual Meeting (Aston, Birmingham).* 1993a.

Haralambous, Y. "Nouveaux systèmes arabes TEX du domaine public". In *Comptes-Rendus de la Conférence « TEX et l'écriture arabe » (Paris).* 1993b.

Haralambous, Y. "Typesetting the Holy Qūran with TEX". In *Comptes-Rendus de la Conférence « TEX et l'écriture arabe » (Paris).* 1993c.

Haralambous, Y. "*Indica*, a Preprocessor for Indic Languages—Sinhalese TEX". In *Proceedings of the 15th TEX Users Groups Annual Meeting (Santa Barbara).* 1994.

Haralambous, Y. and Plaice, J. "First Applications of **Ω**: Greek, Arabic, Khmer, Poetica, ISO 10646/UNICODE, etc.". In *Proceedings of the 15th TEX Users Groups Annual Meeting (Santa Barbara).* 1994.

Knappen, J. "Fonts for Africa". *TUGboat* **14**(2), 104–106, 1993.

Lagally, K. "ArabTEX". In *Proceedings of the 7th European TEX Conference (Prague).* 1992.

Levy, S. "Using Greek Fonts with TEX". *TUGboat* **9**(1), 20–24, 1988.

Nationale, Imprimerie. *Les caractères de l'Imprimerie Nationale.* Imprimerie Nationale, Éditions, Paris, France, 1990.

Plaice, J. "Progress in the **Ω** Project". In *Proceedings of the 15th TEX Users Groups Annual Meeting (Santa Barbara).* 1994.

Vulis, D. "Notes on Russian TEX". *TUGboat* **10**(3), 332–336, 1989.