Methods for Processing Languages with Ω

Yannis Haralambous* John Plaice†

November 10, 1997

Abstract

In this paper we discuss general issues of multilingual typesetting and methods used by the Ω typesetting system. We describe the different levels of language processing with Ω, giving special emphasis to the level of virtual fonts; in particular we give a complete description of the configuration file which is used to create 16-bit virtual fonts out of 8-bit PostScript fonts containing the necessary glyphs.

1 Introduction

Typesetting in different scripts and languages is a problem which has been mainly solved today, either by big software industries which adapt their products to what they consider local typesetting specifications, or by individual users brewing their own limited but practical systems for their personal needs.

Typesetting in different scripts and languages by *k*eeping the *q*uality of traditional typography for every script, and having an open system which can be adapted to any language and script without loss of power or *q*uality, is up to now an unsolved problem, and Ω aims to be a solution to precisely this problem.

If we want to have an open system, and "re-invent traditional typography", for any script and language, just as Don Knuth did with TEX for the English language, we need to analyze and compare the aspects of multilingual typesetting, and establish general methods to solve them.

The purpose of this paper is to show that efficiency and openness can be improved by wor*k*ing on different levels, each one adapted to a specific aspect of multilingual typesetting. These levels correspond to methods used in th Ω system, and we are going to describe them by giving concrete examples from Latin, Gree*k*, Cyrillic, **A**rabic, Hebrew alphabet languages.

*Atelier Fluxus Virus, 187, rue Nationale, 59800 Lille, France. yannis@pobox.com
†plaice@acm.org

2 Ω Methods: an Overview

When developping an Ω typesetting system for a given language, one can work on the following levels:

1. The font level. **A** font is a container of glyphs, needed to typeset in a given language. These glyphs may or may not correspond to the graphical unities of a scripts, weither these are called "letters", "ideograms" or otherwise; these glyphs may be parts of graphical unities, or combinations of them, or new independent symbols.

   They must be provided to the screen previewer and to the printing engine. Ω itself is not concerned by them (just like TEX Ω works only with metrics, and as we will see in the next item, these metrics are not necessarily the ones of the fonts actually containing glyphs).

   The font level is the lowest development level, in the sense that glyphs are indivisible unities which can be used in other higher level structures but cannot be dynamically modified by Ω itself.

2. The virtual font level. **O**nce we have the glyphs we need, we combine them to form what a language usually considers as a gramatically correct script entity (an ring accent alone is of no use, but å is part of the grammar of the Swedish language).

   A virtual font character is a combination of glyphs taken from several "real" fonts, or of other virtual font characters. In the å example above, the glyphs of the ring and of the letter a can actually be taken from entirely different fonts, in different formats (bitmap, PostScript, TrueType, etc.).

   Going from "real" fonts to virtual fonts is mainly a matter of optimisation of storage and memory: taking the sevel Greek vowels, the three accents, two spirits, diaeresis, subscript iota and macron/breve diacritics (that makes 16 glyphs) we produce 336 (!) virtual glyphs. The description of such a character in a virtual font is hounderts of times smaller than the PostScript code describing an hypothetical similar glyph.

   Virtual fonts are the ones used directly by Ω: they can have up to 2^16 positions and 2^32 kerning pairs. The files created by Ω can be processed (previewed, printed) by utilities we have adapted; if the user has to use his/her own utilities which are not "16-bit clean", there is a tool to "devirtualize" the files [i.e. replace virtual fonts by the underlying real fonts] and make them as standard as TEX output files.

3. The ΩTP level. When Ω reads a document, it first tokenizes it and expands commands and then forwards the data to the typesetting engine. Between these two steps we introduce an arbitrary number of filters which we call Ω Typesetting Processes (ΩTPs). These are written in a Lex-like syntax, are

loaded while reading the document and can be activated and de-activated dynamically.

A typical example for an ΩTP is contextual analysis of **A**rabic. **O**f course, this operation can also been done by LaTeX commands; but an ΩTP will do it much faster, it will avoid conflicts with other LaTeX commands, and it will be much easier to create and configure.

Contextual analysis of **A**rabic is a typical example of a script property which is low level and should not use any macros or other high-level structures. In the case of the Latin script one couldn't possibly expect from a user to constantly thin*k* of 'fi' and 'ff' ligatures and place them manually, and it would be very bad strategy to use an "fi ligature command"; in **A**rabic this property is of the same nature, and hence should remain completely transparent.[1]

But efficiency should not only be limited to speed of typesetting**:** sometimes it is very important to optimize also the configuration and customization time and effort. **A** typical example is the management of encodings, weither input or output**:** by using a universal encoding (we call it Unicode++, it is a superset of Unicode) as intermediate step of our ΩTPs, every new input encoding re*q*uires only a foo → Unicode++ ΩTP and every new font encoding only a Unicode++ → foo one; these significantly easier to ma*k*e as if one had to rewrite all processing steps (including contextual analysis and other bells and whistles).

ΩTPs can also be used to hide things from other ΩTPs and ma*k*e them reappear on a later stage**:** imagine you want an **A**rabic word with one letter in red. Placing a `\red` command in front of that letter would normally brea*k* the contextual analysis. Not so if we define a range of "characters" which are considered as characters and not commands, and whose only purpose is to carry information (for example the color switch). These characters would not interfere with contextual analysis, and after the analysis is done, would be transformed into commands.

4. The hyphenation and sorting engines. Hyphenation and sorting rules are grammatical properties of a language**:** they have nothing to do with typographical aspects, input methods, font encodings, etc. They have to be performed on the Unicode++ level, which is the most abstract one. **O**nce defined at this level, they can immediately be used with every input and output encoding. Ω hyphenation and sorting engines are still under development; for the time being, Ω doesn't sort, and hyphenates li*k*e TeX, using the (virtual) font encoding.

5. The macro-command level. LaTeX is a terrific construction, featuring commands on different levels**:** TeX primitives, plain TeX commands, internal

[1] **A**nd undoubtely this would be the case if the lingua franca of computer science was **A**rabic or Urdu, and not English...

LaTeX commands, higher LaTeX commands and environments, user-defined commands and environments. **O**ur goal is to *k*eep all script-dependent processing independent of the LaTeX macro level; once we have reached this goal, every LaTeX pac*k*age will be useable in any script and language, and both Ω pac*k*age and LaTeX pac*k*age developpers will be able to wor*k* independently but still producing mutually compatible software.

In the following sections we will discuss these levels in more detail.

3 Level 0**:** Studying Typesetting **A**spects of the Target Language

This level has little to do with Ω, but it is the most fascinating, and that's way we include it. Learning a language is a difficult process that can ta*k*e years, especially if one is not in the ade*q*uate environment. But studying the typesetting aspects of a given language is a different tas*k*, which is humanly possible in a few months, and which is e*q*ually fascinating. **A**fter having learned the basics (the alphabet/writing system, punctuation, special symbols) one spends months browsing boo*k*s and searching for exceptions, special cases, unwritten laws and conventions, and in general trying to feel the esthetics of a script.

Often one finds symbols un*k*nown even to natives, and almost always, when loo*k*ing in older boo*k*s one discovers what has been lost, what was brought by the computer, what current convention is just a technical compromise and would horrify any traditional typographer.

Once all the facts have been gathered, one has to ma*k*e a model of the language's writing system. For example, traditional Mongolian uses the same model as **A**rabic language (letters connected by a basic stro*k*e, and having contextual forms), only in the vertical direction. Indic languages mostly use the same model**:** letters connected forming clusters, with parts on the left or the right, above or below. Cambodian uses a more elaborate model having consonants in the center of the cluster, subscript consonants centered or adjacent below them, vowels surrounding them, either centered or right justified and eventually accents. Urdu Nastali*q* uses a very particular model**:** characters connected as in **A**rabic but written diagonally from upper right to lower left; furthermore the contextuality of Nastali*q* letters is stronger than the one of ordinary **A**rabic Nas*k*hi or Kufi, since some letters loose their distinctive features (for example, dots) or obtain different features when in the presence of specific other letters [imagine a rule saying "whenever an i is followed by an s or a b, the latter gets the dot of the former**:** ıṡ thıṡ possıḃle?]

After establishing the model one has to decide what is general behaviour of the script and what is exceptional or should always be re*q*uested by the user. Ta*k*e for example the **A**rabic ligature lam-lam-hah ﷲ used in the word **A**llah اللّٰه; this ligature is mandatory for writing the word 'God', and consists of two lams (the second one with shaddah—meaning it is a double lam, so that actually there are three lams—

and vertical fatha), and a hah. When we write these letters without ligature we obtain: ﷲ. We could establish a rule saying "whenever the user asks for three lams, a vertical fatha and a hah, he/she should get the 'God' ligature instead". But is this rule 100% sure? Could'nt we find some other word having exactly those letters, and maybe some other ones? On the other hand, making a macro for the word Allah would be sub-optimal as well. In our transcriptions we have chosen an intermediate approach: using a special way of transcribing this ligature, avoiding any ambiguity with other uses of this character string than 'God'. Of course, this is a solution we have given, the user can easily configure the transcriptions to fit his/her needs, taste and convictions about what is common sense and what not.

Often this modelization is a sequence of more-or-less succesful attempts, hopefully converging to some sufficiently intelligent and compomised system. And this may also change under the burden of real-life use: a typesetting system is not valid until several natives and non-natives haven't done entire books with it, giving the developpers critical feedback, and having created their own transcriptions and conventions.

Enough with generalities, let us return to the different levels of language processing with Ω, starting with the lowest level, which is the font level.

4 Level 1: Dealing with glyphs

What we call a "glyph" is actually simply a character from a PostScript, TrueType or Metafont font; so why calling it that way? Because we want to make the distinction between the "images", and the "combinations of images" which Ω will use as characters for typesetting. In the next section we will see how we combine glyphs; in this section we should normally speak about the design of glyphs, the esthetics of multi-script font design, the aspects of homogeneization, the design of characters which never have been designed before, and the risks of innovation. Discussion of these issues could fill entire books, so we will limit ourselves to a few very precise issues which we had to face when designing these fonts.

4.1 A Multi-script Font: Does it Make Sense?

Does it make sense to design a Cyrillic font with Roman esthetics? Or a Greek font with Cyrillic esthetics? Do these esthetics actually exist? After all we live in a period where most of the printed documents use Microsoft's or Apple's standard Times and Arial/Helvetica fonts which do share the same esthetics—if any—and the first multi-script font would only be the merging of all those fonts.

Let us be realistics: the number of documents using more than one script is very small (although some combinations may occur more often than others: for example, in Greek texts one finds relatively often foreign words written in the Latin alphabet, simply because every Greek can read that alphabet as well) and it certainly a more

serious goal to obtain high *q*uality typesetting staying inside a given script than trying to combine more of them.

But still there are excuses**:** one could try, when going from one script to the other, to *k*eep only those features which can be common without conflicting with a given script's internal esthetics**:** width of stro*k*es (fat/thin, curves, etc.), serifs (whenever appropriate), global dimensions (upper/lowercase letter height and depth, sidebearings, etc.).

One could also try to use common graphical elements, as designer secrets, so that the reader feels a certain homogeneity without exactly where it comes from.

Anyway, to give a direct answer to the *q*uestion, yes, it ma*k*es sense, at least for the Ω project; for two reasons**:** first of all, there is no multi-script font yet (at least not in the public domain) and the typographical atrocities we have seen in the last ten years are already sufficient to motivate us, and secondly because we consider our fonts as starting points**:** we will try to give them the most local typesetting *q*uality without loosing homogeneity—people with more specific (and more limited geographically) needs will replace the parts they consider suboptimal with other fonts, optimal for a given combination of scripts (or a single script) but not for the others. We are giving them the framewor*k* and the tools to do this—and we try to sensibilize them to this eventuality.

4.2 How About Arabic, Hebrew, Indic scripts?

In the previous section we were mainly spea*k*ing about Cyrillic, Gree*k* and Latin**:** three scripts which are close relatives, having common—although far away chronologically—ancestors. When we extend our fonts to Semitic and Indic scripts, some problems vanish and other arise.

The problems that vanish are those of "false twins"**:** a lowercase Latin 'a' is designed to be identical to lowercase Cyrillic one, but is this an universal truth? Must a Cyrillic El 'Л' loo*k* different than a Gree*k* Lambda 'Λ'? The answer seems to be yes (although there are Cyrillic fonts with Gree*k*-li*k*e Els), but how about the Cyrillic Pe vs. Gree*k* Pi? When comparing **A**rabic and Hebrew and Indic and Latin, you don't face those problems.[2]

There is a very interesting counter-example to that assertion; try to identify the script of the following two words**:** Ըus Պաթաաաճեաճի. Does it loo*k* familiar? It should do so, because these glyphs have been designed in absolute conformance to the design principles of Latin script. Guessed what script it is? Well, it is **A**rmenian, and contrarily to our Berber experimentations, this typeface is authentic**:** it can be found in traditionally typeset boo*k*s and doesn't sho*k* **A**rmenians the least. Maybe it started as an experiment a hundred years ago, and has by now become a typeface standard.

But this case remains uni*q*ue**:** although we have seen other scripts designed so that they imitate the Latin script these were only artistic experimentations and can

[2]Except perhaps for punctuation, digits and other general typographical symbols, since more and more scripts tend to use the Latin versions of these.

not be ta*k*en into consideration for general use. Scripts li*k*e the **A**rabic, Hebrew, Indic, South-East **A**sian ones, remain esthetically independent of LGC (Latin-Gree*k*-Cyrillic). What can be done to give a multi-script document, an homogeneous overall impression?

First of all, the fact that lines are shared by different scripts (we suppose here that these scripts are either horizontal only or vertical only), means also that they share the same baseline s*k*ip. This at first glance harmless fact can give very annoying results for scripts with strong ascenders and descenders. For example, Khmer has sometimes double subcripts and accents**:** the baseline s*k*ip must be significantly higher than for LGC; but what if there are only a few Khmer words in a paragraph of several lines of LGC text? Does this mean that only those lines will have a bigger baseline s*k*ip? This is the first problem one has to solve (the solution depends on the combinations of script... there is no general solution).

The second factor to loo*k* at is the width of stro*k*es; many scripts have distinctive fat and thin stro*k*es (not all of them, for example Berber letters are pure geometric forms, drawn with a fixed width pen). If this is the case, one can start by chec*k*ing if these can have the same values as LGC fat and thin stro*k*es. This is of course a very simplistic approach, but it can serve as a good starting point.

We also chec*k* the punctuation is identical, or similar, to the one of LGC (for example, the **A**rabic semicolon is an inverted LGC semicolon, with bigger sidebearings). **A**ll scripts we *k*now of use LGC parentheses, brac*k*ets and braces. These are a good common test to see in which extent a script is different from LGC, and how one can play with the new script to bring it closer to LGC.

If these methods fail, one call always use the—more scientific, and less intuitive—method of comparing typographical "grays"**:** one prepares fonts in different weights, and compares the global impression of gray one gets when loo*k*ing the page from a certain distance, with the one of LGC text. For this it is best to have Metafont fonts, since these allow fine tuning of weight, without redrawing of characters. In fact, through this method, one can find both the right weight and baseline s*k*ip.

4.3 Can **O**ne **A**void Innovation?

Hardly. **A**fter all, we are fixing ourselves new goals**:** our fonts don't have just to be nice-loo*k*ing and to follow a certain number of traditions; we also want the different scripts to fit together, and we also want the typographical variations of one script to have e*q*uivalents in other scripts, or at least we want the entire character set of a script to exist in the same number of variations.

Up to now, an IP**A** font was considered "exotic", and no one really expected it to have bold and italic forms. **A**fter all, as in mathematics bold and italic characters are different symbols, may be IP**A** can have no bold or italic version anyway. This is only partly true**:** some IP**A** characters are in fact "italic characters in upright position", so that globally switching to italic would cause confusion. The same goes for small capitals. Not so for bold, though; a complete Unicode++ font family needs a

bold IP**A** part.

Problems don't end there**:** the IP**A** has been designed to wor*k* with serif typefaces; what about sans-serif ones? The letter 'a' of a sans-serif font may already loo*k* li*k*e an 'ɐ'—this is the case for the Futura typeface—, how do we point out the difference? Same *q*uestion for 'g' and 'ɡ' [one finds the latter even in some serif fonts]. The IP**A** character 'ɪ' (which is in fact a small-cap 'i') differs from the dotless i 'ı' through its serifs, but what if the serifs aren't there? Should we postulate that IP**A** cannot be written with sans-serif fonts? **A**nd what about semi-serif fonts, li*k*e **O**ptima? IP**A** is maybe a very special case, but the same problems arise with **A**frican languages and **A**sian Cyrillic.

And what about Coptic and Slavonic? Just li*k*e ancient Gree*k* can be written with the same typefaces as modern Gree*k* (although there are typefaces specifically made for ancient Gree*k*, li*k*e Porson or Gree*k* Sans Serif), why not typesetting Coptic and Slavonic excerpts without brea*k*ing the homogeneity of the surrounding "modern" font. We have tried to do this for Slavonic (including not only letters and accents found in Unicode, but also all the otyher diacritics and symbols needed for **O**ld Church Slavonic), and the example loo*k*s li*k*e this**:** Ѿвѣща̀ ѐ̋й їѝсъ и́ рєчѐ ѐ̋й. Nothing really exciting to loo*k* at, but that is the point**:** it should fit with the surrounding text instead of loo*k*ing "exotic". This is innovation; is it really useful? Time will show.

4.4 How Much can be Done by Combining Glyphs?

As we will see in the next section, a big part of the Unicode++ table (at least for LGC) is obtained by combining a limited amount of glyphs. This corresponds to a grammatical reality**:** it is *q*uite natural to assume that placing diacritics does not affect the shapes of weither the base character or the diacritic itself. **O**ften this is true, but there are times when typographical *q*uality re*q*uires special shapes. This is typically the example of **A**rabic alif with hamza**:** alif is a "high" **A**rabic letter, we have chosen it to have approx. the height of a capital LGC letter; the hamza diacritic is placed upon the alif; it often happens that an alif with hamza ta*k*es another diacritic**:** a vowel (fatha, or dammah) or even a nounated vowel (fathatan, or dammatan)**:** the latter are already *q*uite high. The combination of alif, hamza and for example, dammatan, would normally be much too high. In traditional **A**rabic typography one can choose smaller diacritics, and after all there many esthetic ligatures which are also *q*uite high, so that the result is homogeneous in the end. But in a modern typeface li*k*e the one in the Ω **A**rabci system, some other solution had to be found. We have chosen to lower the alif letter, so that the alif+hamza has almost the same height as the alif alone**:** compare ا and أ , so that the alif with hamza and a vowel is not excessively high, as can be seen in the following word**:** أُجْرَى .

This method can be applied to several cases in other scripts**:** one can imagine an 'h' with a lower vertical stem, so that the accent of the Esperanto character 'ĥ' is not excessively high. This is finally a matter of taste, and we intend to include this *k*ind of characters as variant forms in our font, and leave the decision on using

them or not to the user.

Another typical example is the one of the diaeresis over Gree*k* vowels iota and upsilon, weither lowercase or uppercase; the distance between the two dots of the diaeresis depends on the letter beneath**:** compare ϊ, ϋ, Ϊ, Ϋ, while in the Latin script the diaeresis (Umlaut, tré) always has the same width**:** ï, ö, ẅ...

5 Level **2:** Combining Glyphs in Virtual Fonts

A character of a virtual font can be a combination of characters of other fonts (weither real, in which case we actually have glyphs, or again virtual, in which case we have sub-combinations etc.), of blac*k* boxes of arbitrary height and width, and of PostScript code [the latter feature is not implemented in all DVI drivers, so one should refrain from using it, at least for the moment].

We have chosen to wor*k* intensely with virtual fonts for two reasons**:** first because by combining glyphs we can optimize space (and space management is crucial when you deal with 16-bit fonts), and second, to be able to use 16-bit fonts without re-writing all DVI drivers of the world (the underlying real fonts are 8-bit only, so that a devirtualized 16-bit font, becomes 8-bit and can be processed by any decent DVI driver).

5.1 The General Scheme

The idea is to have a few "big" virtual fonts, for purposes distinct enough so that one doesn't need to switch fonts all the time. For the moment, the virtual fonts we have are `omlgc` (covering LGC, IP**A**, **A**rmenian, Georgian, Left-to-right Tifinagh) and `omr2l` (convering **A**rabic and Hebrew alphabet languages, Right-to-left Tifinagh and forthcoming Syriac alphabets). **O**ther "big" fonts will follow, for Indic languages, South-East **A**sian ones, CJK and dingbats.

Each "big" virtual font is accompanied by a certain number of "small" PostScript fonts. These have names using the following scheme**:** `om` (for Ω), + `se` or `ss` (for serif or sans-serif) + a two-letter code defining the font encoding (`la` for Latin, `gr` for Gree*k*, `cy` for Cyrillic, `cx` for Extended Cyrillic, and so on) + nothing or `b` or `i` or `bi` (for plain, bold, italic, bold-italic).

To build a "big" virtual font out of these, we use a Perl utility, called `makeovp.pl`. This utility will read a configuration file (which usually has the same name as the font and the `cfg` extension), a special file containing a certain number of "generic" *k*erning pairs, all the **A**FM files re*q*uired to build the font, and a few global parameters ta*k*en from the command line. `makeovp.pl` will build PL files for all PostScript fonts, and an **O**VP file for the "big" virtual font; once this operation is completed, the utilities PLtoTF and **O**VP**2O**VF will convert these files to the binary font files needed by the Ω system for typesetting. This process can be entirely batch (using Unix scripts, D**O**S scripts under Windows or **A**ppleScript on the Mac, with the CMacTeX Ω implementation) and is needed only when the font is modified; an av-

erage user is not expected to modify the font, but nevertheless we give him/her, all the necessary tools to do so.

In the following sections we will describe the configuration file and the operation of `makeovp.pl` in general.

5.2 Files Read by `makeovp.pl`

The Perl utility `makeovp.pl` will read **A**FM files, the configuration file, and the *k*erning file.

Why having a separate *k*erning file, instead of relying on the *k*ernings included in the individual **A**FM files? The answer is trivial**:** because no **A**FM file can contain *k*ernings with characters from other **A**FM files, and this is exactly what we need in our big virtual font**:** for example, punctuation is contained in a "common" PostScript font, while the "Latin" one contains only Latin alphabet letters, the "Gree*k*" one contains Gree*k* letters and specifically Gree*k* punctuation (Gree*k* guillemets), and so on. To *k*ern between Latin letters and common punctuation, or between Gree*k* letters and common punctuation one needs a separate file, with *k*erning pairs from different PostScript fonts.

Otherwise the *k*ernings file has the same syntax as any **A**FM file**:**

```
KPX foo goo -121
```

where foo and goo are characters and -1**2**1 denotes a *k*ern of 1**2**1 PostScript units to the left.

AFM files contain information on character metrics**:** the width of the box containing the character, and the bounding box of the actual shape of the character. This information is sufficient for building a virtual font, and we will see how.

Finally a certain number of global values are given to `makeovp.pl` through the command line, we will see these in the next section, when describing the different operators of the configuration file.

5.3 Structure of the Configuration File

The configuration file (`omlgc.cfg`, `omr2l.cfg`, etc.) contains one line for each character of the virtual font. This line consists of (a) a 4-digit hexadecimal number, which is the character's position in the font, (b) an operator, (c) one or more character names, depending on the operator, (d) a certain number of optional operators and values.

We will describe one by one the different operators, giving concrete examples.

5.3.1 The operator `N`

`N` stands for "N**A**ME", and means simply that the string following the operator is the (PostScript) name of one or more characters in some of the fonts loaded. For example,

```
03A5 N Upsilon
```

means that at position 03**A**5 of the virtual font, we have placed the Gree*k* letter capital Upsilon ϒ. The PostScript names we have used try to be as standard as possible (of course, most of the time there is no standard, so we just have to invent names...)

The same glyph can be used for several font positions**:** for example, the Croatian Dze Đ has exactly the same shape as the Icelandic Eth Đ**:** we will use the same glyph, and hence the same PostScript name. Nevertheless we try to optimize the use of glyphs so that one can typeset in one script without necessarily loading the PostScript fonts for other scripts**:** for example, although the capital Latin '**A**' has teh same shape as the Gree*k* capital **A**lpha, we will use two different glyphs in two different PostScript fonts, so that when typesetting Gree*k* one can avoid loading the Latin PostScript font. These considerations will be unrelevant when we will be able to use 16-bit monobloc PostScript fonts; for the moment, this is not part of the **A**dobe Type 1 font specification.

There are several options we can use with this operator**:** `#KRN`, `#KRNLEFT`, `#KRNRIGHT`, `#HOFFSET`, `#VOFFSET`.

The first three concern *k*erning**:** we can state that a given character has the same *k*erning behaviour as some other character, which we give by name. For example, ç will be *k*erned exactly li*k*e the letter 'c'**:** everytime there is a *k*erning pair in the *k*ernings file using 'c' a new *k*erning pair will be defined, using ç instead of 'c' (and if there is a 'c-c' *k*erning pair, three new *k*erning pairs will be defined**:** 'ç-c', 'c-ç', 'ç-ç'). Sometimes we *k*ern a letter li*k*e some given letter on the right and li*k*e some other letter on the left**:** this is typically the case of ligatures**:** æ will be *k*erned as 'a' on the left, and as 'e' on the right; in that case, we use the `#KRNLEFT` and `#KRNRIGHT` operators**:**

```
00C6 N AE #KRNLEFT=A #KRNRIGHT=E
00E6 N ae #KRNLEFT=a #KRNRIGHT=e
0110 N Eth #KRN=D
```

The operators `#HOFFSET` and `#VOFFSET` will offset the glyph without affecting the box of the character.

5.3.**2** The operators `XHAC` and `CHAC`

These operators will place diacritics over letters, which are considered to have the same height as the lowercase letter 'x' (x-height) or the one of an LGC uppercase letter (cap-height). The idea is that the height of letters can fluctuate**:** a round letter, li*k*e 'o', is slighty higher than a flat letter, li*k*e 'z', to counterbalance a well-*k*nown optical effect. The height of accents over these letters must be the same, even if they aren't exactly of x-height, or cap-height**:** ta*k*e for example ó and ú; if we would ta*k*e the real height of these letters, the accent on the former would be slightly higher than the one on the latter.

How is this accent placed precisely? The Perl utility centers the bounding box of the accent over the bounding box of the letter, with a fixed distance of EPSIL**O**N

(a global value) between the lower boundary of the accent and either the x-height or cap-height of the font (those two are also global values we provide the utility with).

The options available are: `#KRN`, `#KRNLEFT`, `#KRNRIGHT`, `#LETTERLIKE`, `#ACCENTLIKE` and `#LETTERREVLIKE`.

The options `#LETTERLIKE` and `#ACCENTLIKE` allow us to use given glyphs, with the metrics of other glyphs. These options are extremely important in certain cases. **A** typical example is Vietnamese: the letter 'o with hook' ơ is significantly wider than the plain 'o', nevertheless accents have to be centered on the "o part" of the letter: ớ, ờ, ở, ỡ. This trick allows us to correctly place an accent on the vertical stem of a 'b' or an 'h': ḃ, ḣ, ḧ; to obtain this result, we simply ask the accent to be placed as if the letter was an 'l':

```
0603 CHAC b dot #LETTERLIKE=l
0623 CHAC h dot #LETTERLIKE=l
0627 CHAC h dieresis #LETTERLIKE=l
```

We have the same fonctionality with accents: using the `#ACCENTLIKE` operator we can place an accent as if it was some other accent. The typical example is again Vietnamese, where there are combined accents 'circumflex + grave', 'circumflex + acute', which have to be centered with respect to the middle axis of the circumflex (and hence as if there were no acute or grave accent):

```
06D0 CHAC O circumflexacute #KERN=O #ACCENTLIKE=circumflex
06D1 XHAC o circumflexacute #KERN=o #ACCENTLIKE=circumflex
06D2 CHAC O circumflexgrave #KERN=O #ACCENTLIKE=circumflex
06D3 XHAC o circumflexgrave #KERN=o #ACCENTLIKE=circumflex
06D4 CHAC O circumflexhook #KERN=O #ACCENTLIKE=circumflex
06D5 XHAC o circumflexhook #KERN=o #ACCENTLIKE=circumflex
06D6 CHAC O circumflextilde #KERN=O #ACCENTLIKE=circumflex
06D7 XHAC o circumflextilde #KERN=o #ACCENTLIKE=circumflex
```

Another example is Slavonic, with letters such as ѣ, where the accent has to be placed on the right part of the ligature, as in ѣ́. This means that we should use the metrics of a given character, justified on the right of our box: this is the rôle of the `#LETTERREVLIKE` operator.

5.3.3 The operator `VARAC`

This operator has the same syntax (and takes the same options) as the `XHAC` and `CHAC` operators, but produces accents of arbitrary height: it was necessary for letters like 't' which are neither x-height nor cap-height.

5.3.4 The operators `XHLLAP` and `CHLLAP`

These operator have the same syntax (and take the same options) as `XHAC` and `CHAC` but instead of centering the accent over the letter, justify it on the left. We have

used this techni*q*ue for certain cases where the accent placement had to be very precise**:** for the lowercase Gree*k* iota, which ta*k*es special accents (narrower than for the other Gree*k* vowels).

5.3.5 The operators DBLXHAC, DBLCHAC, DBLVARAC, DBLXHLLAP and DBLCHLLAP

These operator have the same properties (and ta*k*e the same options) as the operators without DBL prefix, but ta*k*e two arguments**:** they will place two accents on the same letter, either by centering them or by justifying them to the left. The two accents will be placed an EPSIL**O**N distance apart. **A**n optional parameter #ACCENTOFFSET is used to modify the vertical distance between the accents. We have used this parameter for Gree*k* letters with breve and acute/grave accent, where the latter is brought a little closer to the former than calculated through their bounding boxes (a bounding box gives no information about the actual shape of a glyph).

5.3.6 The operators SUBZERODPAC and SUBFIXDPAC

These operators are used to place diacritics below characters. The first operator will consider the base character being of zero depth and will center the diacritic according to its bounding box, so that the upper boundary of the diacritic is at depth EPSIL**O**N. This is useful for accents such as the lower dot as in ṇ, the lower ring as in a̰, etc.

The second operator will move the diacritic horizontally only, supposing that its depth is already correct. We use this operator for accents such as the cedilla**:** ç, ļ, Ņ, ŗ, etc. Notice that although the cedilla is placed on the virtual font level, the letters with ogone*k* are separate glyphs**:** this comes from the fact that the cedilla is connected to the letter by a straight stro*k*e, which is never tangent to the character, and will neither distort it, nor be distorted by it**:** this allows us to use the same cedilla shape for all character+cedilla combinations. **O**n the other hand, the ogone*k* stic*k*s to the letter and becomes part of its design; every character+ogone*k* combination has been designed separately to give an optimal result.

5.3.7 The operators XHSUBZERODPAC, CHSUBZERODPAC, XHSUBFIXDPAC and CHSUBFIXDPAC

These are combinations of the previous ones, and the CHAC and XHAC operators**:** they allow us to place simultaneously diacritics over and under a character, li*k*e in teh case of Ṩ, ç̌, r̦̄, etc.

5.3.8 The operator ADJ

This operator allows us to concatenate two characters, using a box with width e*q*ual to the sum of the widths of the two boxes, ± the eventual *k*erning between those characters, and height/depth the maximum height/depth of the two characters. We first wanted to use that operator for the Croatian digraphs ‘Lj’, ‘Nj’, etc. but the

decided that the whole idea of having code positions for these digraphs was so silly that we could very well do without. The operator nevertheless proved very useful for cases such as the Gree*k* capital vowels with accent Ἀ, Ἐ, Ἠ, Ἰ, Ὀ, Ὑ, Ὠ.

This operator ta*k*es a special option: `#MOVELEFT`; this option allowed us to override a *k*ern between two characters and move the second one horizontally, together with his box.

5.4 Conclusion

The operators described above allowed us to create virtual fonts for the Latin, Gree*k*, Cyrillic, **A**rabic, Hebrew, **A**rmenian and Berber script. The advantage of the Perl language is that it is platform-independent and that modifications can be *q*uic*k*ly and plainlessly. We will pursue the development of virtual fonts adding new scripts, and adding new operators whenever this is necessary.

Forthcoming developments include operators which will use the PostScript code of given characters to obtain more information: for example the exact coordinates of the vertical stem of letters li*k*e 'h' or 'd', so that accent positioning is more accurate for an arbitrary font.

6 Levels 3 and 4: ΩTPs and LaTeX Macros

Once the structure of fonts is well organized, we use ΩTPs for low level script- or language-dependent operations and macros for higher level operations. It would lead us too far to start discussing general principles of ΩTPs and macros. Instead, we plan to publish a number of papers on the different language and scripts covered by Ω, to start with Multilingual Typesetting with Ω, a Case Study: **A**rabic, which the reader can find in this same volume of proceedings of the 1997 Tsu*k*uba Conference on Multilingual Computing.

To obtain more informations on the Ω project, please visit our Web site:

`http://www.ens.fr/omega`