Mem. A multilingual package for LaTeX with Aleph

Javier Bezos
Typesetter and consultant
http://perso-wanadoo.es/jbezos/
http://mem-latex.sourceforge.net/
jbezos@users.sourceforge.net

Abstract

Mem provides an experimental environment for multilingual and multiscript typesetting with LaTeX in the Aleph typesetting system. Aleph is Unicode-savvy and combines features of Omega and eTeX. With Mem you should be able to typeset Unicode documents mixing several languages and several scripts taking advantage of its built-in OCP mechanism and with a high level interface.

Currently still under study and development, Mem is designed to be capable of following the development of Omega and LaTeX3, and I'm publishing it to encourage other people to think about the ideas behind it and to discuss the advantages and disadvantages of several approachs to the involved problems.

The project is now hosted in the public respository SourceForge.net to open its development to other people.

Introduction

Until now, the only way to adapt LaTeX for it to become a multilingual system is babel; although another systems like mlp (by Bernard Gaulle) or polyglot (by me) have appeared now and then, in practice only babel is used. It exploits TeX in order to accomplish some tasks which TeX was not intended for, like right to left writing and transliterations, but it's clear that the next step requires features not available in TeX. Further, while one can write documents in several languages, babel is esentially a way to change the main language in monolingual documents.

Long ago, Omega and ε-TeX developement started independently and recently a new project named Aleph, combining features from both systems, has been launched. There are several packages for specific languages taking advantage of the features in Omega (devnag, makor, CJK, etc.) and the package omega provided a few macros to ease its use, now expanded with the name of Antomega by Alexej Kryukov [5], but they don't provide a generic high level interface to add a language and to synchronize it with other languages in a consistent and flexible framework. On the other hand, LaTeX3 continues evolving and one of its aims is to have built-in multilingual capabilities.

It is in this context that Mem was born. Actually, it was born several years ago with the name of Lambda and presented in the Fifth Symposium on Multilingual Information Processing (Tokyo, 2001), but for several reasons its development was paused.[1] Its goal is twofold: in the short-term, to provide a real working package for Aleph to become useable with LaTeX, taking advantage of features like the OCP mechanism; in the mid-term, to use the experience gained with a real life system in order to develop better multilingual environments with LaTeX3 and Omega.

The rest of this paper of devoted to highlight some of the issues and therefore it does not intend to be exhaustive. To get a full picture of the package please refer to the manual [3], which is being written at the same time as the package, because I think the documentation is an integral part in the development process. I've divided the topics in two parts, those related directly to TeX, and those related to the Aleph/Omega extensions, particularly to the OCP mechanism.

The TeX part

Organizing and selecting features Language commands are grouped in *components*, with a few predefined ones—namely, names, date, tools and text. At first sight this resembles babel, but in fact this similitude is only superficial, because you are free to organize and to select components. The limit

[1] There is no paper, but you can find the slides on http://perso.wanadoo.es/jbezos/mlaleph.html. In fact, Mem was born even before, in 1996, with the name of polyglot as I shall explain shortly.

would be a component per macro but this does not seem sensible; for example, left and right guillemets could be a single group. On the other hand, too many components would be unconvenient for the user. I think a sort or component/subcomponent model should be devised (eg, `text.guillemets`), and at the time of this writing I'm working on a system to allow even decisions at macro level like `text.guillemets.\lguillemet`.

This poses the problem to determine which components are active at a certain point of the document. There are, of course, systems like those in CSS and other formatting languages based on description rules for transfomations based on content (for example, with the keywords `inherit` and `ignore`). However, TEX allows programmable rules for transformation based on format and such a model seems very limited (and the term "inherit" can be inappropiate in the context of an object-like environment).[2] Unlike CSS, with its closed set of properties, TEX allows creating new properties and therefore new ways to organize the document layout.

There is a proposal from Frank Mittelbach and Chris Rowley [7] based on nesting levels, with comments about the main issues to be addressed, but since this paper is somewhat abstract regarding the possible solutions it's difficult to determine if that model will be enough for many purposes. In particular, it presumes the structure of the document is a tree, and therefore, as its authors point out, the model has to be extended to provide the necessary support of "special regions" that receive content from other parts of the document.

A basic idea in that paper is that there is a base language for large portions of text as well as embedded languages segments, which are nestable. Although in a limited way, these concepts shown at TUG 1997 related to a clear separation between base and embedded languages were present at that time in my own polyglot package (first released early 1997) whose code I used as the base to develop Lambda and now Mem.

On the other hand, Plaice and Haralambous in [9] and I (in Lambda) proposed independently to follow a model based in context information; the versioning system for Omega described in the former has been worked out and much extended from a theoretical point of view in [11] by Plaice, Haralambous and Rowley, with the introduction of the concept of a *typographical space*. Unfortunately, such a model cannot be carried out in full with TEX and it has not been implemented in Omega, but to me it's clear it should be taken as a guide for Mem, and for that matter for any multilingual environment. At the time of this writing I was studying how to tackle this task and the resulting model will be left for a future paper.

[2] See [2]. An English summary is availaible on `http://mem-latex.sourceforge.net`.

Never again default values! In a well-known article published in the TUGboat ten years ago, Haralambous, Plaice and Braams proclaimed "Never again active characters!" [4]. Now I proclaim the end of another source of problems in the babel package—namely, default values. Actually, default values are mainly associated with active characters, but they are also present in macros. Having default values for a certain language is not a bad thing, but when those values are restored every time the language is selected and they cannot be redefined with the standard LATEX procedures then problems arise.

In Mem, a default value in a language is only a proposal, while the final decision is left to the user, which can change it by means of `\renewcommand`, `\setlenght` and similars. No special syntax is required, like for example `\addto\extrasspanish`. The behaviour of language commands is exactly that of normal commands, except that their values change when the language changes.

A macro is made specific for a certain language with `\DeclareLanguageCommand`, which provides a default definition to be used if the users likes it; if you don't like it, you can redefine it, since the default value is not remembered any more. Outside that language, there could be macros with similar names, but they are not language specific (except if defined for another language, of course).

Furthermore, if a language defines an undefined macro, this is only defined in the context of that language and you not are required to provide a default for *another* language, because I firmly believe loading a language should not change at all the behaviour of another language. In other words, with Mem languages are much like black boxes.

A good example could be the Basque language, which places the figure number before the figure name. For that to be accomplished we must make Basque dependent several internal macros. Considering the number of languages and the fact we cannot know *a priory* which changes will be necessary, the fact languages can (or even must) decide which macros have a default value could lead to an unmanageable situation which could even prevent a proper writing of packages, because we don't know if we need to use `\(re)newcommand` or something else.

The Aleph/Omega part

OTP files The OCP mechanism provides a powerful tool to make a wide range of text transformations which are not possible with preprocessors. Since OCPs perform transformations after expanding macros, we can guarantee all characters, and not only that directly "visible" in the document, are taken into account. One of the main aims of Mem is to develop a high level interface for them, because using the Omega primitives is somewhat awkward. Moreover, since OCPs must be grouped in OCP-lists before actually applying them, the advantages of a high level interface becomes aparent—OCP-lists are hidden to users and language developers and they are built and applied on the fly depending on the language and the context, thus avoiding the danger of a combinatorial explosion [11, p. 107]. For further details on how OCPs works, see the Omega documentation [8] and the very useful case study [10].[3]

A key concept in Mem is that of *process*, a set of OCPs performing a single logical task. Very often, a task cannot be carried out by just one OCP, but in more complex cases a set of interrelated OCPs will be necessary. A very good example of this is the devnag package for Omega by Yannis Haralambous, where mapping from Unicode to the target font requires three OCPs. At the time of this writing I'm working on OCPs to handle the Latin/Cyrillic/Greek family of scripts, which is being a lot more involved as one could think at first sight, and very likely a set of three OCPs will be necessary to carry out the single process of mapping from Unicode to the T1, T2*n* and LGR encodings.[4] This is particularly true for Greek with its many possible ways to represent the many possible combinations of letters and accents, which is far from trivial.[5]

[3] Still, the former is very technical and the latter is very basic, and unfortunately an "intermediate" manual explaining the implications of OCPs is not available yet, thus meaning developing OTPs must be done very often by trial and error. The Aleph Task Force and I are considering the possibility to write such a manual.

[4] In addition, it should be investigated if several of the tasks done by these OCPs can be delegated to a virtual font.

[5] And the LGR encoding has some odd assignments, like placing GREEK PSILI AND OXIA at `"5E` (^) thus having the catcode of superscript. There is another symbol mapped to the backslash. That would not be important except for a long-standing bug in how OCPs treat catcodes which the Aleph Task Force is trying to fix, because it's a critical one. Since there are very few LGR fonts, and very likely their number will not increase, I'm thinking about removing the support for that encoding and instead to write a virtual file. To add further confusion, the Omega standard font `omlgc` moves the Unicode Greek Extended chars to a non standard placement.

It's important to remember where OCPs are not applied: when writing to a file (e.g., the `aux` file), in `\edef`'s, in arguments of primitives like `\accent`, and in math mode. The latter is a serious limitation, and the Aleph Task Force is working on a solution. This means Mem has done very little in these areas, except redefining `\DeclareMathSymbol` to allow higher values.

Extending OTP syntax: MTP files Perhaps the main limitation of OTP files, containing the source code of OCPs, is that the only letters we can use are those in the ASCII range, while for the rest of the Unicode range we must use numerical values. MTP files have been devised to overcome these limitations so that we can use Unicode names instead of numbers (see figure 1). Currently, they are converted to OCP with a little script named mtp2ocp, a preprocessor written in Python.

Another addition to OTPs is that it maps spacial characters to several points in the Private User Area whose catcodes are fixed (as defined by the Mem style file). This way, characters like `\`, `{`, `$`, etc., have the expected behaviour even in verbatim mode.

I hope MTP files could help in the near future to make the task somewhat simpler, so suggestions are most welcome. This way we can have prototypes to experiment with, so that in the future otp2ocp itself could be extended with new features if necessary. (One of the reasons I use Python is that it's a great language for prototyping.)

Unicode as input encoding Unicode, unlike many other encodings, clearly separates characters and glyphs. This means that at character level, Unicode can introduce controls to provide further information about these characters, including how they should be rendered. It is expected that this information has to be processed in order to decide which glyph to use. Traditional font formats (TrueType and PostScript) do not have this capability or it is limited.

Unicode, considered as an input encoding, is quite different from other encondings and poses several challenges which must be taken into account if we want to read properly Unicode text. Currently, conversions done by LaTeX packages or Omega OCPs just ignore these controls and instead it is supposed the user must supply them with TeX macros.

For example:[6]

- letters with diacriticals, either composed or decomposed,

[6] For some hints on that, see [13]

```
.....................
[LATIN CAPITAL LETTER L WITH STROKE]     => <= @"8A ;
[LATIN SMALL LETTER L WITH STROKE]       => <= @"AA ;
[LATIN CAPITAL LETTER N]{botaccent}<0,>[COMBINING ACUTE ACCENT]
                                         => <= @"8B \(*+1-1);
[LATIN SMALL LETTER N]{botaccent}<0,>[COMBINING ACUTE ACCENT]
                                         => <= @"AB \(*+1-1);
[LATIN CAPITAL LETTER N]{botaccent}<0,>[COMBINING CARON]
                                         => <= @"8C \(*+1-1);
[LATIN SMALL LETTER N]{botaccent}<0,>[COMBINING CARON]
.....................
[LATIN SMALL LETTER I WITH MACRON]
        => <= [LATIN SMALL LETTER I][COMBINING MACRON];
[LATIN CAPITAL LETTER I WITH BREVE]
        => <= [LATIN CAPITAL LETTER I][COMBINING BREVE];
.....................
[CENT SIGN]            => "\UseMemTextSymbol{TS1}{162}";
[POUND SIGN]           => "\UseMemTextSymbol{TS1}{163}";
[CURRENCY SIGN]        => "\UseMemTextSymbol{TS1}{164}";
[YEN SIGN]             => "\UseMemTextSymbol{TS1}{165}";
.....................
<acc> [COMBINING GRAVE ACCENT]        => "\UseMemAccent{t}{0}";
<acc> [COMBINING ACUTE ACCENT]        => "\UseMemAccent{t}{1}";
<acc> [COMBINING CIRCUMFLEX ACCENT]   => "\UseMemAccent{t}{2}";
```

Figure 1: Several chunks from MTP files using Unicode names. Currently symbols are hardcoded, not an ideal situation.

- ligatures marked with ZERO WIDTH JOINER,[7]
- hyphens, non breaking hyphens, non breaking spaces, etc.,
- fixed width spaces,
- variation selectors,
- byte order mark.

In order to unify the character encoding used in style files, only utf-8 and explicit Unicode values (eg, `^^^^0376`) are used, but that poses the problem with a non-Unicode document since changing the OCP for the input encoding would mean kerning and ligatures are killed. To overcome this well known TEX limitation, input OCPs use an internal switch mechanim to escape temporarily to utf-8 or utf-16 (see figure 2). The trick is to pass information to the OCP with the character `^^1b`, whose meaning in many character encodings is `ESCAPE`, followed by another character with the operation to be performed. I'm not sure if this mechanims is robust enough, but if it were the idea could in the future serve as a way to pass context information to a certain OCP so that its behaviour may be changed, although of course a built-in mechanism as that proposed by John Plaice *et al.* [11] would be preferable.

[7] The semantics of this character has been extended in Unicode 4.0 and now can be used to mark ligatures [12, p. 389ss]

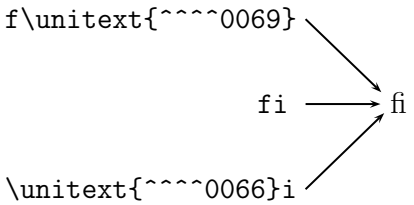

Figure 2: Entering a Unicode character with Mem does not break ligatures.

LATEX Internal representation This section is devoted in part to a few ideas which I put forward in the LATEX3 list, which was followed by a very long discussion about a multilingual model (or more exactly, multiscript) for LATEX. These ideas lead to introduce the concept of LICR (LATEX internal character representacion). Actually, LATEX has for a long time had a rigorous concept of a LATEX internal representation but it was only at this stage that it got publicly named as such and its importance realised.[8] The reader can find more on LICR in the second edition of *The LATEX Companion*, by Frank Mittelbach and others [6, section 7.11.2].

What LICR does is essentially to ensure there is only a way to represent a certain character so that

[8] Chris Rowley, "Re(2): [Omega] Three threads", e-mail to the Omega list, 2002/11/04.

different input methods (say, á and \'{a}) lead to the same representation (in that case \'a) and that this representation is able to find a correct glyph somehow.[9] The required funtionality for that to be accomplished is splitted in two well know packages—namely, inputenc and fontenc.

As far as I know, no paper explaining the technical details of the LICR has been published, so I'm going to attempt an operational definition. Before doing that, I think remembering different kinds of TEX expansion process is to the point (I exclude one level expansion as done by \expandafter):

- \def no expansion.
- \edef expands anything except non expandable tokens.
- protected \edef expands anything except non expandable tokens *and* protected tokens (even if expandable).
- execution expands anything and performs the actions of primitives.

So, we can say LICR is what we get in a protected expansion.

Unicode provides this kind of "internal representation" but without the normalization of LICR. Let's remember Unicode allows representing characters with diacritics in composed form (eg, ä) or in decomposed form (eg, a¨), and that these forms *may* be normalized to either composed or decomposed forms. There are three possibilities:

- normalizing to composed forms.
- normalizing to decomposed forms.
- not normalizing at all.

Decomposition has, in turn, several types, but we won't discuss them in this paper.

The questions here are: Is it possible the preserve the LICR in Mem?; if so, must be the LICR preserved in Mem? Does it fit in the Unicode model?

In order to answer these questions, we must remember the LICR relies heavily in active characters, which will be replaced in Mem by OCPs. Furthermore, macros are expanded and executed (see above) before OCPs are aplied thus making impossible any attempt to catch things like \'a. It seems that an alternative method to inputenc/fontenc must be provided.

Once we have an expanded string, characters are normalized to decomposed characters instead of the composed form favoured by the Web Consortium, for example (it should be noted that in the LICR letters are decomposed). The reasons are

[9] Note the LICR is not necessarily a valid input method, because \'a is not always correct in LATEX.

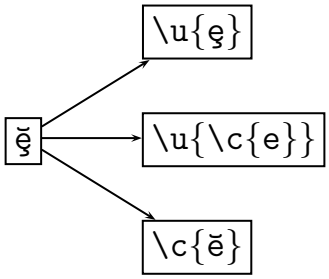

Figure 3: Several ways to input the same character. With Mem the four are strictly equivalent, because they are converted to Unicode and normalized. With the NFSS, if ę does not exist, then the ˘ is always faked. However, with Mem, if ę does not exists but ě does, then ¸ is added to the real composite character.

mainly practical, because the composed form to be selected in some cases depends on the glyphs available. Since normalizing to composed forms would require decomposing, sorting diacriticals and then composing, and font processes would require decomposing again and sorting again to see if there are matching glyphs for the first accent above or the first accent below (or even a combination of both), by using directly the decomposed form we are avoiding a lot of overhead (see figure 3). In fact, the Unicode book says [12, p. 115]:

> In systems that *can* handle nonspacing marks, it may be useful to normalize so as to eliminate precomposed characters. This approach allows such systems to have a homogeneous representation of composed characters and maintain a consistent treatment of such characters.

This dual representation of characters is what is making processes for the Latin/Cyrillic/Greek script so complex, but we have to deal with them if we want a Unicode typesetting engine.

The Latin script has a rich typographical history, which not always can be reduced to the dual system character/glyph. As Jaques André has pointed out, "Glyphs or not, characters or not, types belong to a class that is not recognized as such" [1]. Being a typesetting system, neither Aleph nor Mem can ignore this reality, and therefore we will take into account projects like the Medieval Unicode Font Initiative (MUFI)[10] or the Cassetin Project. However, it doesn't mean a Unicode mechanism will be rejected when available. For example, ligatures can be created with the ZERO WIDTH JOINER. If there

[10] http://www.hit.uib.no/mufi/

is a certain method to carry out a certain task in Unicode, it will be emulated.

Diacritical marks The Unicode 4.0 book states [12, p. 184] when discussing spacing modifier letters:

> A number of the spacing forms are covered in the Basic Latin and Latin-1 Supplement blocks. The six common European diacritics that do not have encodings there are added as spacing characters.

In other words, except for these six diacritics (U+02D8-U+02DD), the spacing forms of combining characters are those in the range U+0000-U+00FF. Unfortunately, it happens this is not true, since the spacing caron accent (U+02C7) is not encoded in these blocks. Further, one of these six diacritics encoded separately—namely, the tilde U+02DC—does exist in these blocks (U+007E).

What to do, then? One will be forced to find some kind of hint, and one can do it readily—all characters in the block Spacing Modifier Letters are prefixed with MODIFIER LETTER, except the six spacing clones and CARON (U+02C7). From this, we can infere that the right spacing form for the circumflex accent is not the MODIFIER LETTER variant, but the one in the Basic Latin Block, exactly like the ACUTE ACCENT. No doubt the "small" tilde has been encoded separately because the ASCII tilde has already a special meaning in several OS's.

Still, I think there is a better solution, or rather a better encoding which does not pose this problem. Since the glyphs for diacritics are mainly intended for use with the `\accent` primitive, one can conclude they are, after all, combining characters. The fact we need further processing with TEX does not prevent considering these glyphs conceptually as non-spacing characters—this is just the way TEX works. Since composing diacritical marks are encoded anew in Unicode, we don't need to be concerned with legacy encodings and their inconsistencies.

Conclusions

In this paper I have scratched only the surface of some topics, which deserve by themselves a whole paper. In addition, many others have not been even treated like for example:

- Hyphenation, including patterns for Unicode-like fonts.
- Automatic selection of languages and fonts depending on the current script.
- Since letters are not active any more, one should be allowed to write `\capítulo` or `\κεφάλαιο` instead of `\chapter`.
- Fonts—monolythic or modular?
- OpenType—must its information be extracted so that it's under our control? (However, using OpenType fonts with TEX is still a failed subject, although there are interesting projects like XeTEX.[11])

Before finishing this paper, I would like to cite Frank Mittelbach in a message posted to the LATEX3 list:

> The fact that we don't agree with some points in it only means that the processes are so complicated that we haven't yet understood them properly and so need to work further on them.

I hope Mem will provide an environment which would help us (including me) to understand better how OCPs work as well the issues a multilingual system poses.

References

[1] André, Jacques: "The Cassetin Project – Towards an Inventory of Ancient Types amd the Related Standardized Encoding", *Proceedings of the Fourteenth EuroTEX Conference*, Brest (France), 2003.

[2] Bezos, Javier: "De XML a PDF, tipografía con TEX", *Proceeding of the IV Jornadas de Bibliotecas Digitales*, Alicante, Spain, 2003 [in Spanish].

[3] Bezos, Javier: "Mem: A multilingual environment for Lamed/Lambda", 2004, `CTAN: macros/latex/exptl/mem/mem.pdf`

[4] Haralambous, Yannis, John Plaice and Johannes Braams: "Never again active characters! Ω-Babel", *TUGboat*, Volume 16 (1995), No. 4.

[5] Kryukov, Alexej: *Typesetting Multilingual documents with Antomega*, 2003, `TeXLive2003: texmf/doc/omega/antomega/antomega.pdf`.

[6] Mittelbach, Frank, and Michel Goossens: *The LATEX Companion,* Addison-Wesley, 2nd ed., 2004.

[7] Mittelbach, Frank, and Chris Rowley: "Language Information in Structured Documents: A Model for Mark-up and Rendering", `http://www.latex-project.org/papers/language-tug97-paper-revised.pdf`.

[8] Plaice, John, and Yannis Haralambous: "Draft documentation for the Ω system", 2000, `TeXLive2003:/texmf/doc/omega/base/doc1-12.ps`.

[11] `http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=XeTeX&_sc=1`

[9] Plaice, John, and Yannis Haralambous: "Supporting multidimensional documents with Omega", Fifth International Symposium on Multilingual Information Processing, Tokyo, Japan, 2001, `http://omega.enstb.org/papers/dimensions.pdf`.

[10] Plaice, John, and Yannis Haralambous: "Multilingual typesetting with Ω, a Case Study: Arabic", `TeXLive:/texmf/doc/omega/base/torture.ps`.

[11] Plaice, John, *et al.*: "A multidimensional approach to typesetting", *TUGboat,* Volume 24 (2003), No. 1.

[12] The Unicode Consortium: *The Unicode Standard, Version 4,* Addison-Wesley, 2003.

[13] The Unicode Consortium: *Unicode in XML and other Markup Languages*, Unicode Technical Report #20, W3C Note 13 June 2003.