Mem
A *M*ultilingual *e*nviron*m*ent for Lamedh/Lambda

Javier Bezos-López

October 19, 2004

Note: This is a draft. Don't expect all things explained here will work, but many of them work!

1 Introduction

The package Mem provides a language selection system for Aleph/Omega taking advantage of the features of LaTeX 2$_\varepsilon$. It provides some utilities which make writing a language style quite easy and straightforward and its aims are mainly:

- to provide a set of high level macros for users and developpers of language styles, which "hide" the involved primitives and make them easier to use.
- to coodinate different languages so that Aleph/Omega will become a true multilingual environment.

Some of the features implemented by Mem are:

- You can switch between languages freely. You have not to take care of neither head lines nor toc and bib entries—the right language is always used.[1] You can even get just a few commands from a language, not all.
- Dialects—small language variants—are supported. For instance American is a variant of English with another date format. Hyphenations patterns are attached to dialects and different rules for US and British English can be used.
- Customization is quite easy—just redefine a command of a language with `\renewcommand` when the language is in force. The new definition will be remembered, even if you switch back and forth between languages. This way, Mem essentially suggest while the final typographical decisions, which are a matter of style, are left to you.*Note: And this idea must be further extended.*
- An unique layout can be used through the document, even with lots of languages. If you use a class with a certain layout you don't want to modify, you or the class can tell mem not to touch it at all.
- Mem understands Unicode composite character, at least to some extent. A file using decomposed characters (eg, a ̂ instead of â) will be typeset as expected.[2]
- Integrated tools to type trasliterated text, so that yo can say *dobryj den'* and it will be transliterated to the Cyrillic alphabet. That will allow to enter the text with the Unicode characters used in transliterations, too (eg, Sharḥ Ibn ʻAqīl ʻalá Alfīyat Ibn Mālik).

[1] Index entries follow a special syntax and the current version of mem cannot handle that. For this reason the index entries remain untouched and you may use language commands only to some extent.

[2] Actually, `\^` is redefined to create a composite character.

2 Quick start

Once installed, you can use mem. To write a, say, German document simply state the `german` option in the `\documentclass` and load the package with
`\usepackage[charset=isolat1]{mem}` (if you are using the latin1 encoding; if the document uses the Unicode encoding this setting is not necessary). That's all. If you are happy with that you need not go further; but if you are interested in advanced features (how to insert a Spanish date, for instance) just continue.

3 User Interface

3.1 Components

A language has a series of commands and variables (counters and lengths) stored in several components. These components are:

names Translation of commands for titles, etc., following the international LaTeX conventions.

layout Commands and variables for the general layout of the document (`enumerate`, `itemize`, etc.)

date Logically, commands concerning dates.

tools Supplementary command definitions.

These components belong to two categories: names and layout are considered *document* components, since they are intended for formatting the document; date and tools are considered *text* components, since you will want to use them temporarily for a short text (or may be not so short) in some language. There is a further component which is hidden to users: processes, which contains macros used by traslation processes dealing with text typographical conventions and transformations.

3.2 Selecting a Language

You must load languages with `\usepackage`:

```
\usepackage[english,spanish]{mem}
```

Global options (those of `\documentclass`) will be recognized as usual. You can cancel this automatic selection with the package option `loadonly`:

```
\usepackage[german,spanish,loadonly]{mem}
```

`\languageset*[`*no-components*`,`*properties*`]{`*language*`}`

Selects all components from *language*, except if there is the optional argument. *no-components* is a list of components *not* to be selected, in the form `no`*component*`,no`*component*`,...`. For instance, if you dislike using tools:

```
\languageset*[notools]{french}
```

Note: perhaps this syntax based on no-components should be changed This command also sets defaults to be used by the unstarred version (see below). Properties change the behaviour of some features of the language.

`\begin{`*language*`} ... \end{`*language*`}`
`\begin{languageset}[`*components*`,`*properties*`]{`*language*`} ... \end{languageset}`

Where *components* is a list of *component*s and/or `no`*component*s.

The simplest way to use a language locally is the environment above. The second one provides a more flexible approach.

There are three posibilities:

- When a component is cited in the form *component* (e.g., `date` or `names`), this component is selected for the current language, i.e., that of the current `\languageset`.
- When a component is cited in the form `no`*component* (e.g., `nodate` or `nonames`), this component is cancelled.
- When a component is not cited, then the default, i.e., that set by the `\languageset*` in force, is used; it can be a `no`-form, and then the component will be disabled if necessary. If there is no a previous starred selection, the `no`-form will be presumed in all of components.

If no optional argument is given, then `text,tools,date` are presumed. Thus, at most two languages are selected at the same time; this point should be stressed, because it means that you cannot use at the same time features belonging to three or more languages. While at first glance this behaviour may seem very limiting... *Note: to be filled Note: to repeat, perhaps the way components are selected should be changed, but I have not found a simpler syntax.*

Here is an example:

```
\languageset*[notools]{spanish}
Now we have Spanish names, date, and layout,
but no tools at all.
\languageset[names,nodate]{french}
Now we have Spanish layout, French names,
but no tools.
\languageset[tools]{german}
Now we have Spanish names, date, layout,
and German tools.
```

Selection is always local. There is also the possibility to use `languageset` as command and `\languageset*` as environment.

`\begin{languageset}*[`*no-components*`]{`*language*`}` *text* `\end{languageset}`
`\languageset[`*components*`]{`*language*`}`

It very unlikely that you will use the starred version at all.

For the sake of clarity, spaces are ignored in the optional argument, so that you can write

```
\begin{languageset}[date, tools, no text]{spanish}
```

`\`*language*`text{`*text*`}`
`\languagetext[`*components*`]{`*language*`}{`*text*`}`

A short text in another language. The behaviour of some features could be different. Use this command inside paragraphs (i. e., in horizontal mode) and `languageset` between paragraphs (i. e., in vertical mode—`\languagetext`, unlike `\languageset`, does not change the shapes of paragraphs.

`\languageunset`
`\languagereset`

You can switch all of components off with `\languageunset`. Thus, you return to the original LaTeX as far as mem is concerned.[3] You can return to the status before the last `\languageunset` with `\languagereset`.

A typical file will look like this

[3]Well, not exactly. But you should not notice it at all.

```
\documentclass[german]{...}
\usepackage[spanish]{mem}
\newenvironment{spanish}%
  {\begin{quote}\begin{languageset}{spanish}}%
  {\end{languageset}\end{quote}}
\begin{document}
Deutscher text
\begin{spanish}
  Texto en espa'nol
\end{spanish}
Deutscher text
\end{document}
```

Note that `\languageset*{german}` is not necessary, since it is selected by the package. (Because there is no `loadonly` package option.) Note also that the *language* enviroments can be redefined in terms of `languageset` (but `languageset` must not be modified).

3.3 Package options.

- `charset=`*charset* Sets the input encoding for the whole document.

3.4 Properties

Properties change the behaviour of some language features. Some properties are available in all languages, while some others are specific to certain languages. The first group includes (all take a value):

- `charset` The input encoding if different from the encoding of the rest of the document.
- `rmfamily`, `sffamily`, `ttfamily` Set the font family for the language. If not given, Mem uses the script values, or else the current font.
- *encoding* Overrides the default font encoding list in the cfg file. It can be a single value or a list enclosed in braces.
- `hyphenation` Overrides the default hyphenation (currently patterns should be loaded when generating a format). *Note: To be implemented*

`\languageproperties{`*language*`}{`*properties*`}`

The properties set in the preamble with this command will be added automatically when the *language* is used.

3.5 Tools

`\languagename`

The name of the current language. You must not redefine this command in a document.

`\languagelist`

Provides a list of the requested languages.

```
\allowhyphens
```

Allows further hyphenation in words with the primitive `\accent`.

```
\nofiles
```

Not a new command really, but it has been reimplemented to optimize some internal macros related with file writing.

```
\languageensure
```

The mem package modifies internally some LaTeX commands in order to do its best for making sure the current language is used in a head/foot line, even if the page is shipped out when another language is in force. Take for instance

(With `\languageset*{spanish}`)

```
\section{Sobre la confecci'on de t'itulos}
```

In this case, if the page is broken inside, say, a German text, an implicit `\languageensure` restores `spanish` and hence the accents.

Yet, some non-standard classes or packages can modify the marks. Most of times (but not always) `\languageensure` solves the problem:

(With `\languageset*{spanish}`)

```
\section{\languageensure Sobre la confecci'on de t'itulos}
```

In typeset, writing and other modes it's ignored.

```
\unichar{code}
\unitext{text}
\utftext{text}
```

Conversion tools. The first one is a character with Unicode position *code*. The argument of the second and third macros is text to be preserved as two byte Unicode encoding, or transcoded from utf-8, respectively. Ligatures are preserved with `\unichar`, `\unitext` and `\utfstring`, but not with `\utftext`. *Note: See yatest.tex. The names must be cleaned up.*

3.6 Scripts

Writing systems are automatically handled by mem, so you should not be concerned too much with that. But there are some interesting points to be noted:

- If the document uses a single encoding, no matter it is Unicode or, say, ISO 8859-6 (Latin/Arabic), it will do its best in order to switch automatically the language when the script changes. The order which the languages are loaded in `\usepackage` is important, because the last language loaded using a certain script will be the default language for that script and used when an untagged change of script comes across.*Note: How that should be implemented? Is it possible?*

- Fonts can be attached to a certain script. Mem stores the current font attributes and reselects the font (automatically selecting the font encoding) when the language changes, if desired. For example,

  ```
  \scriptproperties{El}{rmfamily = grtimes}
  ```

  Note: Still under study and development

4 Developper Commands

Some command names could seem inconsistent with that of the user commands. In particular, when you refer to a language in a document you are referring in fact to a dialect, which belogs to a language. As user, you cannot access a language and you instead access a dialect named like the language. From now on, when we say "current language" we mean "current language or dialect." For more details on dialects, see below.

4.1 General

`\DeclareLanguage{`*language*`}`

The first command in the `.ld` must be this one. Files don't always have the same name as the language, so this command makes things work.

`\DeclareLanguageCommand{`*cmd-name*`}{`*component*`}[`*num-arg*`][`*default*`]{`*definition*`}`
`\DeclareLanguageCommand*{`*cmd-name*`}{`*component*`}[`*num-arg*`][`*default*`]{`*definition*`}`

Stores a command in a component of the current language. The starred version makes sure that the utf-8 encoding is used, thus overriding the document encoding. The definition will be activated when the component is selected, and the old definition, if any, will be stored for later recovery if the component is switched off.

There is a point to note (which applies also to the next commands). Suppose the following code:

At `spanish.ld`:
```
\DeclareLanguageCommand*{\partname}{names}{Parte}
```

At document:
```
\languageset*{spanish}
\renewcommand{\partname}{Libro}
\languageset*{english}
\partname
```

Obviously, at this point `\partname` is 'Part'. But if you follow with

```
\languageset*{spanish}
\partname
```

Surprise! *Your* value of `\partname`, i.e., 'Libro', is recovered. So you can customize easily these macros in your document, even if you switch back and forth between languages.

`\SetLanguageVariable{`*var-name*`}{`*component*`}{`*value*`}`

Here *var-name* stands for the internal name of a counter or a lenght as defined by `\newconter` (`\c@...`) or `\newlenght`. The variable must be already defined. When the component is selected, the new value will be assigned to the variable, and the old one will be stored.

`\SetLanguageCode{`*code-cmd*`}{`*component*`}{`*char-num*`}{`*value*`}`

Similar to `\SetLanguageVariable` but for codes. For instance:

```
\SetLanguageCode{\sfcode}{text}{`.}{1000}
```

Languages with `\frenchspacing` should set the `\sfcodes` with this command, so that a change with `\nonfrenchspacing` is recovered after a switch.

`\UpdateSpecial{`*char*`}`

Updates `\dospecials` and `\@sanitize`. First removes *char* from both lists; then adds it if it has categorie other than 'other' or 'letter'. With this method we avoid duplicated entries, as well as removing a character being usually special (for instance `~`).

4.2 components

`\DeclareLanguageComponent{`*component*`}`
`\DeclareLanguageComponent*{`*component*`}`

Adds a new component. With the starred version, the component will be considered a `text` component, and hence included in the default of `\languageset`. Component names cannot begin with `no` because of the `no`-form convention given above.

4.3 Translation processes

In the context of Mem, OCP's/OTP's become *processes*. However, a single conceptual process can be splitted into several OCP files because it requires more than one step. There are several levels of processes, each of them perform some specific task. The order which processes are applied and their names are determined by the following commands.

`\DeclareLanguageProcess{`*id*`}{`*level-name*`}`

Declares a slot where ocp's could be added. You won't use this command very often, except if the four basic components—namely, charset, input, text, and font—don't fit your needs. *Note: A more "abstract" syntax could replace id by a aftername which name is added after (numerically).*

`\AddLanguageProcess{`*level-name*`}{`*ocp files*`}`

Adds the stated ocp's to the given slot for the current language.

`\UseLanguageProcess`

Activates the ocp's corresponding to the current language, including those declared with `\DeclareLanguageProcess` but excluding the generic processes described below.

`\DeclareMemProcess{`*id*`}{`*level-name*`}`

Translation processes not attached to languages, but used as generic tools.

There are five processes predefined in Mem, some of them in the kernel and some others in script definition files:

charset (200) Converts the input text to Unicode (language).

unicode (400) Apply Unicode transformations if necessary, such as normalization (language). *Note: Which one to be used is still under study—composed or decomposed?*

transcript (600) Transliterate from one script to another, for example with `charset=isolat1, input=latin` for Cyrillic (language). *Note: Is this the right order?* .

input (800) Input conventions like TEX pseudo-ligatures such as `---` (language).

case (1000) Case changes and similar transformations within a script (like Japanese katakana/hiragana) (Mem).

text (1000) Language dependant processes to follow typographical conventions (letter variants in Greek, spacing with puntuation marks in French, contextual forms in Arabic, etc. Language)

font (1200) Transcoding to the target font and faking missing characters (like accented letters. Mem).

`\AddMemProcess{`*level-name*`}{`*ocp file*`}`

Defines the stated ocp and creates a process with level *level-name* and the same name as the ocp file.

`\AddMemProcess{`*level-name*`}[`*process-name*`]{`*ocp files*`}`

Defines the stated ocp and creates a process with level *level-name* and the name *process-name*. To be used if the process consists in several ocp files.

`\UseMemProcess{`*ocp name*`}`

Activates the translation process corresponding to the *ocp name* (= *ocp file*).

4.4 Scripts

All languages has at least an attached writing system which is written in; information for scripts is generic to languages using it (for instance, what `\guillemetright` or `\'` means). Each script has an associated file with extension `.sd` and named with the two letter codes from the ISO 15924 standard (lowercased, in the arguments you should use the mixed case of the standard).

`\SetLanguageScript{`*language-code*`}`

Every language file should contain a command like that. It loads the macros corresponding to the language (diacriticals, punctuation, etc.) and performs some additional task.

The following macros can be used in the `.sd` files.

`\DeclareScript{`*language*`}`

Set up.

`DeclareScriptCommand{`*name*`}{`*definition*`}`

Declares a macro whose definition which will be in force with a certain script. `\DeclareScrinpCommand` is essentialy a disguise for `\DeclareTextCommand` because the

internal handling of script macros is essentially the same than LaTeX 2ε font encodings. For example, `la.sd` (latin) contains

```
\DeclareScriptCommand{\~}[1]{#1\unichar{"0303}}
```

while `el.sd` (greek) contains

```
\DeclareScriptCommand{\~}[1]{#1\unichar{"0342}}
```

`\SwitchScript{`*language*`}`

You should not use this macro. It is inserted automatically by Aleph/Omega when it thinks that a change of script is necessary.

`\UseMemAccent{`*cmd*`}{`*position*`}{`*letter*`}`

Places the accent in *position* over *letter*. The *cmd* parameter is just a remainder of the accent (`\'`, `\"`, etc.) This command is intended to be used in otp files, and only to give support to fonts not compliant with mem (namely, 8t, 8r, 7t, etc. fonts).

4.5 Dates

`\DeclareDateFunction{`*date-function-name*`}{`*definition*`}`
`\DeclareDateFunctionDefault{`*date-function-name*`}{`*definition*`}`
`\DeclareDateCommand{`*cmd-name*`}{`*definition*`}`

By means of `\DeclareDateCommand` you can define commands like `\today`. The good news is that a special syntax is allowed in *definition* with date functions called with <*date-funtion-name*>. Here <*date-funtion-name*> stands for the definition given in `\DeclareDateFunction` for the current language. If no such function for the language is given then the definition of `\DeclareDateFunctionDefault` is used. See `english.ld` for a very illustrative example. (The `<` and `>` are actual lesser and greater signs.)

Predeclared functions (with `\DeclareDateFunctionDefault`) are:

- <d> one or two digits day: 1, 2, . . . , 30, 31.
- <dd> two digits day: 01, 02, . . .
- <m> one or two digits month.
- <mm> two digits month.
- <yy> two digits year: 96, 97, 98, . . .
- <yyyy> four digits year: 1996, 1997, 1998, . . .

Functions which are not predeclared, and hence should be declared by the `.ld` file, are:

- <www> short weekday: mon., tue., wes., . . .
- <wwww> weekday in full: Monday, Tuesday, . . .
- <mmm> short month: jan., feb., mar., . . .
- <mmmm> month in full: January, February, . . .

The counter `\weekday` (also `\value{weekday}`) gives a number between 1 and 7 for Sunday, Monday, etc.

For instance:

```
\DeclareDateFunction{wwww}{\ifcase\weekday\or Sunday\or Monday\or
  Tuesday\or Wesneday\or Thursday\or Friday\or Saturday\fi}
\DeclareDateCommand{\weektoday}{<wwww>, <mmmm> <dd> <yyyy>}
```

4.6 Dialects

As stated above, `\languageset` access dialects rather than languages. `\DeclareLanguage` declares both a language and a dialect with the same name, and selects the actual language.

`\DeclareDialect{`*dialect*`}`
`\SetDialect{`*dialect*`}`
`\SetLanguage{`*language*`}`

`\DeclareDialect` declares a dialect, which incorporates all declarations for the current actual language. With `\SetDialect` you set the dialect so that new declarations will belong only to that dialect. `\DeclareDialect` just declares but does not set it.

A dialect with the same name as the language is always implicit. You can handle this dialect exactly as any other dialect. In other words, after setting the dialect, new declarations belong only to it. If you want to return to the actual language, so that new declarations will be shared by all dialects, use `\SetLanguage`.

Note that commands and variables declared for a language are set by `\languageset` before those of dialects, no matter the order you declared it.

For example:

```
\DeclareLanguage{english}
\DeclareDialect{american}
Declarations
\SetDialect{english}
\DeclareDateCommmand{\today}{...}
\SetDialect{american}
\DeclareDateCommand{\today}{...}
\SetLanguage{english}
More declarations shared by both english and american
```

4.7 Interaction with Classes

`\mem@no`*component*
(i.e. `\mem@nonames`, `\mem@nolayout`, `\mem@notext`, etc.)

Initially, these commands are not defined, but if they are, the corresponding components are not loaded. This mechanism is intended for classes designed for a certain publication and with a very concrete layout which we don't want to be changed. You simply write in the class file

```
\newcommand{\mem@nolayout}{}
```

Note this procedure does not ever load the component—if you select it nothing happens.

5 Configuration

The way languages and scripts are referred to inside mem is highly customizable. You can refer a language using the name in its own language (and even its local script), the english name or

even the name in your own language. For that to be accomplished a set of configuration files are provided. *Note: Currently, only that with English names are provided. Note: To be implemented. However, I think that this feature leads to unmanageable configurations; I think that it should be restricted to English and local names*

`mem.cfg`

The languages in English. Every line must contain three fields:

name The name to be used in the document; the name of the language/dialect as used in the `.ld` file.*Note: A solution to that could be to introduce the possibility to define "synonymous" in the document*

file The file name, which uses the three-letter codes from ISO-639 and the extension `.ld`.

patterns The hyphenation patterns to be used with the language/dialect.

6 Customization

"Well, but I dislike `spanish.ld`." You can customize easily a language once loaded, with new commands or by redefininig the existing ones.

- If want to redefine a language command, simply select the component of this language which defines it (as with `\languageset*`) and then redefine it with `\renewcommand`.
- If you want to define a new command for a language, first make sure no language is selected (for instance, with `\languageunset`). Then `\SetLanguage` and use the declaration commands provided by the package and described above.

A further step is creating a new file, by copying it, modifying the commands and, of course, renaming the file! Or with a file with extension `.ld` as:

```
\ProvidesFile{...ld}
\input{...ld}  the file you want customize
\languageset*{...}
Commands to be redefined
\languageunset
\SetLanguage{...}
Commands to be created
```

You can modify languages by means of a package. The `spguill` is an example.

7 Errors

`Unknown component`

You are trying to assign a command to an inexistent component. Perhaps you have misspelled it.

`Missing language file`

Probable causes:

- Wrong configuration.
- The corresponding `.ld` file is missing or misplaced.

`Unknown language`

You forgot requesting it. Note that dialects stand apart from languages; i.e., you have no access to `austrian` just requesting `german`.

`Invalid option/property skipped`

In the starred version of `\languageset` you must use the `no`-forms only.

`Bug found (`*n*`)`

You will only find this error when using the developpers commands—never with the user ones—or if there is a bug in description files written by others. In the latter case, contact with the author. The meaning of the parameter n is

1. *Unknown component in declaration.* The component hasn't been declared. Perhaps you misspelled it.
2. *Invalid component name.* Component names cannot begin with `no` to avoid mistakes when disabling a component.
3. *Declaration clash.* You are trying to redeclare a command or to set a new value to a variable for this language. If you want redefine it, select the language and simply use `\renewcommand` (or `\set..`). If you intend to define a new command for the language, sorry, you must change its name.
4. *Invalid language/dialect setting.* Generated by `\SetLanguage` or `\SetDialect` when the argument is not a declared language/dialect.

Now we describe how mem generates TEX and LATEX errors because of intrinsic syntax problems.

`TeX capacity exceeded, sorry [save_size=`*n*`]`

You are using to many ungrouped languages. Fix: use the environment version of `languageset`, the *language* environments, or alternatively use `\languageunset` before a new `\languageset`.

8 Miscelaneous

This section is devoted to some miscelaneous topics which will be put in the right context once this documentation is more complete.

8.1 Mathematics

Aleph/Omega extends the possibilities of math fonts by enlarging the range of possible glyphs to 65.535. However, currently OCP's are not applied in math mode ... [to be filled]

Mem redefines `\DeclareMathSymbol` to accept large values (ie, to use `\omathcode` and `\omathchardef`).

8.2 Accents

Mem understands both composed and decomposed diacritical marks (at least in simplest cases) and can normalize them so that they are properly displayed—this way you can write `\'{\c{\"{a}}}` and the accents are rearranged and stacked.

Unicode allows two ways to represent accented letters: either composed (ie, *ġ* is a single character whose code is U+0121) or decomposed (ie, *ġ* is two caracters, *g* followed by ˙ with code U+0307).

In order to get an internal representation as close as Unicode as posible, the accent commands are redefined as in the next example:

```
\DeclareScriptCommand\`[1]{#1\unichar{"0300}}
```

so that the text can be handled uniformly. After normalizing to the composite form, Mem leaves to the font encoding process decide if the char exists in the font or if it should be decomposed and then faked with the help of `\accent` and related commands.

8.3 Encodings

Currently, the following encodings are supported:

- T1/TS1. Almost complete, but still with some parts missing.
- OT1/TS1. Incomplete.
- ULA. Omega Unicode-like for Latin.
- UEL. Omega Unicode-like for Greek.
- UCY. Omega Unicode-like for Cyrillic.
- UAR. Omega Arabic. It uses cuni2oar, which apparently mixes contextual analysis and font enconding—to be investigated

  Note: The previous encodings are uncomplete and very problematic. Together, they are like UT1, for Latin/Greek/Cyrillic with names because of two reasons: UT1/omlgc is a sort of "modified Unicode" and therefore not Unicode at all, ut1cmr exists but pointing to a OT1 enconded font, and ot1omglc points to a UT1 encoded font (!). Further, should a single enconding contain the full Unicode set?
- T2A/TS1. Incomplete.
- LGR. It uses uni2lgr, which maps Latin characters to Greek glyphs—to be replaced with a correct OCP.

Note: Currently, font processes point directly to the target glyph. Another possibility is to load the LaTeX enconding then then use the LICR name, which in turn has the glyph code (which is, in fact, the procedure described below=. Pros of the latter: better control from within TeX; cons: font encodings has to be preloaded. I think it would be a nice thing if people has not to be concerned with font encondings (and if possible, we should minimize input encondings, perhaps giving a platform/language default so that [linux, czech] will be enough?

8.4 Private User Area

Mem uses the first page of the Public User Area for special purposes in the following way:

- `\uE000-\uE00F` are reserved for characters having always catocodes in the range 0-16. Actually, some of them does not make sense (eg, 11 and 12).
- `\uE020-\uE0FF` duplicates the ASCII range but making sure they are not special characters.

8.5 The original idea

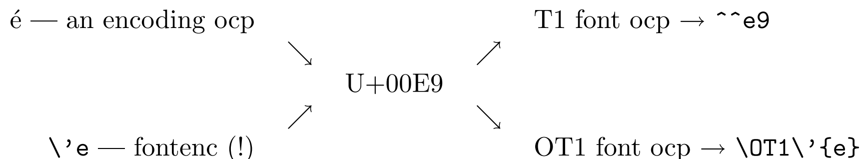
This section is devoted to a few ideas which I put forward in the LaTeX3 list, which was followed by a very long discussion about a multilingual model (or more exactly, multiscript) for LaTeX. These ideas lead to introduce the concept of LICR (LaTeX internal character representacion). Actually, LaTeX 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.[4] The reader can find more on LICR in the second edition of *The LaTeXCompanion*, by Frank Mittelbach and others (section 7.11.2).

Let's explain how TeX handle non ascii characters. TeX can read Unicode files, as xmltex demostrates, but non ascii chars cannot be represented internaly by TeX this way. Instead, it uses macros which are generated by inputenc, and which are expanded in turn into a true character or a TeX macro by fontenc:

é — inputenc → `\'{e}` — fontenc → `^^e9`

That's true even for cyrillyc, arabic, etc., characters!

Omega can represent internally non ascii chars and therefore actual chars are used instead of macros (with a few exceptions). Trivial as it can seem, this difference is in fact a *huge* difference. For example, the path followed by é will be:



It's interesting to note that fontenc is used as a sort of input method!

For that to be accomplished with ocp's we must note that we can divide them into two groups: those generating Unicode from an arbitrary input, and those rendering the resulting Unicode using suitable (or maybe just available) fonts. The Unicode text may be so analyzed and transformed by external ocp's at the right place. Mem further divides these two groups into four (to repeat, these proposals are liable to change):

1a) charset: converts the source text to Unicode.

1) input: set input conventions. Keyboards has a limited number of keys, and hands a limited number of fingers. The goal of this group is to provide an easy way to enter Unicode chars using the most basic keys of keyboards (which means ascii chars in latin ones). Examples could be:

   - `---` → em-dash (a well known TeX input convention).
   - ij → U+0133 (in Dutch).
   - no → U+306E [the corresponding hiragana char]

Now we have the Unicode (with TeX tags) memory representacion which has to be rendered:

2a) writing: contextual analysis, ligatures, spaced punctuation marks, and so on.

2b) font: conversion from Unicode to the local font encoding or the appropiate TeX macros (if the character is not available in the font).

[4] Chris Rowley, "Re(2): [Omega] Three threads", e-mail to the Omega list, 2002/11/04. I've discovered recently an article by Robin Fairbains advancing some of the ideas in Mem ("Omega – why bother with Unicode", *TUGboat* 16/3, 1995) such as the clear separation in the functions of ocp's, which has been applied, for example, to devnag after presenting Mem (by then named Lambda) in Tokyo 2001.

Since before step 2 we have a Unicode representation, we can process the text with external tools compatible with Unicode (using `\externalocp`; an interface to this feature must be added in the near future). This would be useful for, say, Thai word boundaries.

This scheme fits well in the Unicode Design Principles, which state that that Unicode deals with memory representation and not with text rendering or fonts (with is left to "appropiate standars").

There are some additional processes to "shape" changes (case, script variants, etc.).

8.6 MTP files

Mem has a new kind of OTP file named MTP. It extends the OCP syntax, by mean of a preprocessor written in Python, to provide the following features:

1. Special characters mapped to the Private User Area in order to have characters with the right catcode in verbatim. Mem sets these catcodes accordingly.

2. Characters inserted with their Unicode name enclosed with brackets, like `[COMBINING CARON]` which is lot more readable than `@"030C`.

This little tool, whose code is somewhat simple-minded, will be extended to allow UTF-8 characters. Binaries for Windows will be created, too, but a conversion to C would be nice, I think; very often, Unix and Linux have built-in Python interpreters.

8.7 Verbatim

Verbatim text with OCP's is a nuissance, because unlike macros replacements does not save the catcodes of characters. [—To be filled—]. *Note: verbatim makes Aleph/Omega to enter sometimes in a infinite loop, but until now I have not discovered why. Unfortunately, OTP's even recatcode letters, so that something like `\string^` does not work as expected— ^ is recatcoded to 'math superscript'!*

8.8 Math

Mem redefines `\DeclareMathSymbol` to accept large values (ie, to use `\omathcode` and `\omathchardef`).

8.9 Extensions to Unicode

The Latin script has a rich typographical history, which not always can be reduced to the dual system character/glyph. As Jaques Andr has pointed out, "Glyphs or not, characters or not, types belong to a class that is not recognized as such."[5] 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) or the Cassetin Project.

However, it doesn't mean Unicode mechanism will be rejected when available. For example, ligatures can be created with the ZERO WIDTH JOINER. If there is a certain method to carry out a certain task in Unicode, it will be emulated, like for example glyph variant selectors *Note: Really??.*

9 Final remarks

Note: There are some areas which I have not stydied in depth, particularly wrinting directions. I have some ideas, but they must worked out.

[5] "The Cassetin Project," *Proceedings of the Fourteenth EuroTEX Conference,* Brest, 2003.