The $\Omega$ Typesetting and Document Processing System

Yannis Haralambous & John Plaice

1 A bit of history

In the dark ages of computing (around 1970) a man called Donald E. Knuth, professor at Stanford University, began writing what was to become one of the fundamental texts on computer programming: *The Art of Computer Programming*. Like any other writer he wanted his books to be well typeset; but unlike any other writer he decided to also develop the necessary tools to achieve this. What he expected to be a one-year undergraduate student project finally took him ten full years to finish. He called this project $\TeX$: a programming language for typesetting, the appropriate compiler, the necessary fonts, utilities, etc. Almost thirty years later, $\TeX$ is still used, and together with “industry” technologies such as the PostScript language, it makes a very powerful typesetting tool. Nevertheless there is one particularity of the $\TeX$ project: it is frozen, in the sense that Knuth decided in 1990 to stop developing $\TeX$. People can still contribute by writing code in the $\TeX$ programming language but the $\TeX$ compiler will stay *ad aeternum* as it was in 1990.

This would be a minor issue if everyone would use it only for writing English math and computer science books such as the *The Art of Computer Programming*. Obviously this is not the case. When one tries to apply $\TeX$ to other contexts it quickly shows its limits. One example is typesetting in other languages than English. It is a general phenomenon that the world of computing is English-language-oriented (probably because most computer development and marketing is done in the United States). Things that are considered to be natural in other languages (such as writing right-to-left or top-to-bottom, or combining diacritics, or having large sets of characters) become a real problem when they need to be done by English-oriented software.

Many people, including the authors, have tried to circumvent $\TeX$’s deficiencies by writing complicated $\TeX$ code or by applying external preprocessors and postprocessors. These techniques have proven that $\TeX$ is able to do everything, but the results are clumsy and unnatural. It soon became clear that a new system was needed to further extend $\TeX$, and solve specific problems.

This is why the authors have been building, since 1994, a new system, based on $\TeX$, in which to explore new areas such as multilingual typesetting and
XML document processing. The system is called \( \Omega \), the last letter of the Greek alphabet, often understood in Western culture, as the summit, the ultimate or the supreme.

One of \( \Omega \)'s goals is to be as natural as possible for every language, as if the whole system was conceived with that language in mind. For example, one can write in any direction in the same way, with the same properties and functionalities. Large sets of characters are of no difficulty to \( \Omega \) either, since internally it is 16-bit (and soon 31-bit). Combining diacritics are dealt with by a very low-level process, so that it remains compatible with higher-level \( \TeX \) programming code written by the user, but at the same time is under the total control of the user. And these are just a few examples.

Since 1994 until today, the authors have been steadily developing and adding new features to \( \Omega \). Finding solutions for issues as fundamental to humanity as writing systems is a fascinating experience. One of the most interesting challenges is that programming requires a certain amount of rigor, which is not always the case for writing systems. A typical example is the Thai language: as in many Far East languages, sentences are made of words which are not visually separated. Words are made out of syllables. When breaking a paragraph into lines, a break may occur between syllables of the same word, or between words: only in the first case is a hyphen needed to be added by the typesetting system. Hence, the typesetting system must know the word limits. Any Thai speaker knows the word limits of his/her language but asking him/her to mark these word limits while keying in Thai text would be against the naturality principle of \( \Omega \): if people have longstanding typing habits, then this is what they consider as “the most natural way” of keying in their language; \( \Omega \) must make it possible to them to continue using those habits. Thus, we have to teach \( \Omega \) how to recognize Thai word limits, even in texts where these have not been marked. This is a linguistic issue, involving morphological analysis of the text.

Working on \( \Omega \) involves computer science, linguistics, typography, font design, history of language and other disciplines. The success of the Unicode encoding has proven to be very useful for the \( \Omega \) project: not only Unicode provides \( \Omega \) with a proper basis to work upon, but it also has sensitized the entire computing community on the need for internationalization. It is \( \Omega \)'s goal to cover more and more languages/parts of the Unicode encoding and hence to become a de facto typesetting system for Unicode texts. But one should not confuse “typesetting system” and “rendering engine” — and this confusion occurs unfortunately more and more often. Unicode is an information interchange encoding and there is a fundamental difference between interchange of data, and [typographical] representation of the same data, between content and form, between character and glyph. This difference, however fundamental, is rather theoretical in the case of the English language (where there is almost a one-to-one correspondence between characters and glyphs), but becomes more and more important for other languages and writing systems. Not to mention the fact that because of the increasing use of American desktop publishing software for typesetting all around the world, typography also tends to be uniformized, and local typographical traditions vanish at the same speed as the Amazon forest.
Ω is sufficiently open and powerful to be adapted to any context wherever in the world: indeed, being a public domain project it transgresses barriers of economical nature. There is a long way to go to implement typographical traditions of all languages of the world, and that’s where Ω is heading to.

2 Fonts

The central idea of typography is the use of “movable type”, that is the decomposition of a text into signs (letters, punctuation, etc.) which are repeated, combined, placed next to each other to form lines, and the lines placed vertically to form pages. A font is a collection of such objects. The computer font paradigm follows corresponds to the hot metal types of Gutenberg’s press (or of wooden types, used in China, some centuries before Gutenberg).

\[ \text{\LaTeX} \] has attempted to implement this paradigm by introducing the following principles:

- typography is two-dimensional;
- a book is made out of pages, a page out of lines, a line out of words and punctuation, a word out of letters;
- at first, a letter is just a box: this box has two vertical dimensions (height and depth, with respect to the baseline) and one horizontal (width);
- at a second stage (which actually occurs outside of \[ \text{\LaTeX} \]), these boxes are filled with shapes, either in the form of pixels or of Bézier curves;
- letters interact in two ways: they can attract or repel each other (“kerning”) or can be transformed into new signs, called “ligatures”.

\[ \text{\LaTeX} \] fonts are made following these principles. This approach would be sufficient if there were not other constraints that have appeared in the years since \[ \text{\LaTeX} \] was developed. One of the constraints is encoding, that is the correspondence between a character (letter, digit, punctuation, etc.) and its ordinal number in the font (we say: its “position”, since fonts are most often displayed as tables with each character defined by its coordinates). When \[ \text{\LaTeX} \] was developed, D. E. Knuth chose an encoding called ASCII for a major part of each font. This was a wise decision, since ASCII is still around; but ASCII, like Unicode, is an information interchange encoding, and hence by definition cannot be sufficient for typography. \[ \text{\LaTeX} \] fonts had additional signs, which although not in ASCII, were necessary for the needs of typography. This, original, \[ \text{\LaTeX} \] font encoding is now called ‘OT1’.

In the last thirty years, hundreds of different encodings have been defined: some were font encodings like OT1, defined by various font designers, computer manufacturers, software companies, etc., other were information interchange encodings, some of them official, others private. Many encodings were defined for specific needs of computer users in non-English speaking countries.
The problems started when \TeX\users wanted to use fonts from other origins than \TeX, or, inversely, when they wanted to use \TeX\ fonts in other programs (for example, graphical design software). The latter problem has never really been solved (in fact, other means have been found to produce \TeX-like expressions in other contexts, such as the \texttt{psfrag} package which replaces text found inside EPS files by text typeset by \TeX). The former problem has led to an interesting new concept: the one of virtual fonts. Virtual fonts are "abstract fonts", in the sense that characters are only empty boxes and do contain neither pixels nor Bézier curves. They are used as intermediate steps between \TeX\ and fonts from the outside world, which can be of arbitrary encoding.

Virtual fonts not only re-encode existing fonts, they can also combine glyphs taken from different fonts into "composite" characters. This can be very useful, for example when accented letters are needed: a "real" font (one with real pixels or Bézier curves) can contain letters and accents, and the virtual font can combine them into accented letters.

More problems arise when one tries to do "multilingual" typography, in the sense that a given system has to produce equally appropriate results in different linguistic contexts. A typical example is the one of the dieresis (= \texttt{\textipa{e}} = Umlaut) accent placed upon Latin letters: in German typography the accent is lower than in French typography. In cases like that, a "multilingual" font has to satisfy multiple and often contradictory constraints. There are two obvious solutions: either make the font big enough so that it covers all possible cases, or make it dynamic, in the sense that some properties of characters will change depending on the context. In other words, either the font is big enough, or it is intelligent and can produce the necessary combinations on-the-fly.

The example of the dieresis accent is only a tiny problem compared to the needs of many non-Latin alphabet languages, where glyphs are combined forming more and more complicated clusters. Japanese furigana are such an example: most of the time one or two furigana placed above (or to the right of, if typesetting is done vertically) an ideographic character are enough, but sometimes there is a multitude of furigana that must be placed above/to the right of a group of characters in a very specific and well-defined way. In the case of \TeX\ this problem would be solved by considering all of the characters (base characters and furigana) as separate "words", and combining them using two-dimensional translations. This is only partly efficient, since text loses its unity and becomes a gathering of individual boxes containing characters.

Another deficiency of \TeX\ becomes apparent when typesetting Japanese or other CJK languages: \TeX\ fonts have been designed to only be used horizontally. One can of course typeset vertically using a standard \TeX\ font, but the (vertical) distances between letters will not be optimal. In vertical typesetting, CJK accents (bullets, etc.) are placed on the right or on the left of base characters, centered according to horizontal axes. This means that the action of "accenting" (or "diacriticizing" from the Greek verb \texttt{diakritos} = to distinguish), always uses axes orthogonal to the main direction of text. These axes are not necessarily splitting the letter in half: for some non-symmetrical glyphs, the axes of accents may very well be de-centered. In other words, to do decent multilingual and
multi-directional typography one needs to know, for each character of each font, the position of each axis, whether vertical or horizontal (see fig. 1).

![Diagram of multi-directional typography](image)

Figure 1: Illustration of primary and secondary baseline and accenting axes. In the case of the Latin script, primary baseline \( p \rightarrow p' \) is horizontal, primary accenting \( \bar{p} \rightarrow \bar{p}' \) is vertical bottom to top; secondary baseline \( s \rightarrow s' \) is vertical top to bottom and secondary accenting \( \bar{s} \rightarrow \bar{s}' \) is also vertical, but bottom to top. In the case of ideographic script, primary baseline \( p \rightarrow p' \) is vertical top to bottom, primary accenting \( \bar{p} \rightarrow \bar{p}' \) horizontal left-to-right, secondary baseline \( s \rightarrow s' \) is horizontal left-to-right and secondary accenting \( \bar{s} \rightarrow \bar{s}' \) vertical bottom to top.

By browsing the Unicode standard book, one realizes very quickly that diacritics placed above/beneath or right/left of a character are not the only ones needed: in some cases, such as the Hebrew dagesh, diacritics are even placed inside a glyph. Not to mention the various ways of combining diacritics, of the same kind or of different kinds.

Placing diacritics on characters is a nice kind of sport and indeed some languages have many more diacritics than others (for example, Vietnamese). But diacritics should not affect other aspects of the character's value and behavior: for example, as long as the diacritics do not protrude, letters with or without diacritics should be kerned exactly the same way: compare ÁVÁTÁR (badly kerned) and ÁVÁTÁR (well kerned). Other example: in a grammar book, some letters in some words may be underlined to emphasize a given grammatical rule; these words should be hyphenated exactly in the same way as if there were no underlined letters.

To solve these problems, Ω introduces new principles to the fundamental
paradigm of \TeX:

- fonts shall be big enough to contain all necessary glyphs for typesetting;

- a letter is still just a box; the triplet of values already used in \TeX (height, depth, width) will be the information corresponding to the main text direction; the box carries additional information: height, depth and width corresponding to the secondary text direction, symmetry axes for main and secondary direction, and the possibility to add extra information (additional coordinates of points) for special cases, like accents placed inside the glyph, etc.;

- typography is done at multiple levels (page > line > word/letter > accent > accent of accent) and each time we change levels, we also switch directions (from primary to secondary, and back). All properties letters had in \TeX for the primary direction, will have their analogs in the secondary direction.

- whatever happens on level \(n + 1\) shall not affect typography of level \(n\), unless there is a problem of protrusion.

Here is an example, to illustrate the last principle: a page is made out of lines; let us call this "level 1". Each line is made out of words; let us call these "level 2". Whether these words use letters with ascenders or not, the distance between lines is always the same (called "leading"). This distance is always the same, except in the—unfortunate—case when a word contains a letter such big that it would overlap with the previous line if the leading was kept fixed. Placing accents on letters leads us to "level 3". As said before, the existence of one or more accents on a letter should not affect its kerning with other letters, except if these accents may overlap with the surrounding base letters, or with their accents. This means that one must be able (a) to ignore level \(n + 1\) when typesetting level \(n\), (b) to control whether level \(n + 1\) may cause overlapping when doing (a), (c) to modify the behavior of level \(n\) when such overlapping occurs.

The reader is invited to re-read the previous paragraph and realize that although it describes to some extent the structure of a page, it does not explicitly mention if the page is typeset vertically or horizontally. In fact, as already said in the section about directions, \(\Omega\) treats them symmetrically, all directions having the same amount of functions and properties.

This new model of typography (subdividing a document into levels, and changing direction at every level change), will allow \(\Omega\) to solve problems of however complex typography, be it in the context of Asian languages, natural or artificial languages with many diacritics, or even other not necessarily linguistic contexts where pages are filled with two-dimensional constructions (musical notation, technical drawings, etc.).
3 Writing in different directions

One of the key assumptions in most text processing software is that writing takes place horizontally, with text flowing left-to-right, with successive lines flowing top-to-bottom. This form of writing is used by much of the world’s population, but by no means by all.

Arabic, Hebrew and several other scripts originating in the Middle East are written horizontally, from right-to-left. Traditional writing and printing in East Asia, including Japan, is done vertically, with the first line of the page on the right hand side. Uighur and Mongolian writing is also vertical, but the first line of the page is on the left.

To complicate matters, it is possible for different scripts, which have different “natural” directions, to be intermixed on the same page, thereby creating some situations where a script may be typeset in a rotated manner, in order for it to fit in a broader context.

Omega solves all of these problems in their full generality. It assumes that a writing direction can be designated by three characters, where each is one of Top, Bottom, Left, and Right. These characters absolutely designate one of the edges of the physical page. Then a writing direction must designate:

Primary part. The “top” of each page.

Secondary part. The “left” of each page.

Tertiary part. The “top” of each character.

The secondary direction must be orthogonal to the primary direction. The tertiary direction can take all four values. Hence there are 32 possible directions. Here are the most common ones:

TLT — Left-right scripts, horizontal CJK.

TRT — Right-left scripts.

RTT — Vertical CJK, upright left-right scripts in vertical CJK.

RTL — Mongolian in vertical CJK.

RTR — Rotated left-right scripts in vertical CJK.

LTL — Mongolian.

LTR — Rotated left-right scripts in Mongolian.

LTT — Vertical CJK in Mongolian.

There are four direction parameters. they are

- \pagedir dir. The direction of the current page, as used for headers, footers, footnotes, and such things.
\bodydir\ dir. The direction of the main part of the page, i.e. the real
text.

\pardir\ dir. The direction of the current paragraph.

\textdir\ dir. The direction of the text within the current paragraph.

\mathdir\ dir. The direction of mathematics when it appears.

THERE WILL BE AN EXAMPLE HERE.

4 \Omega TP s

One of the design principles of \Omega is to make typesetting as natural as possible,
in every typesetting context. As we already saw in the introduction, for historical
reasons, “natural” typesetting often means that \Omega provides the intermediate
steps between typewriter-like input (or “logical” input) and typography. For example,
in French typography, a semicolon is always preceded by an unbreakable
thin space (this space is flexible: it is at most a thin space, that is more-or-less
a fourth or a sixth of 1 cm, but must always be less than the interword space,
which itself is flexible). When keyboarding French text, people are used to type
an empty space in front of the semicolon. Hence, the “most natural” way of
typesetting would be to input the text with blank spaces in front of semicolons
and to obtain, in the output, the appropriate thin space, as described above.

It is obvious that the definition of a thin space, according to the few rules
given above, requires a certain number of \TeX commands, which can be compressed
into one macro-command, but not less than that. Hence, \Omega has to insert
this macro-command in front of every semicolon (except of course for semicolons
that are part of mathematical formulas, or that are not in a context of French
language).

To make such a transformation easier, \Omega introduces the concept of \Omega TP = \Omega
Translation Process. An \Omega TP is a kind of a filter that reads the text (more
precisely, the data stream) and takes appropriate actions, according to the context.
Another frequent case is the one of uppercase/lowercase transformation. In all
Latin, Greek, Cyrillic and Armenian alphabet languages there is the concept of
upper and lower case letters. The typesetting system sometimes has to convert
text from one case to the other, for example, some books have their headings
written in upper case. This transformation is highly untrivial: first of all, one
must know exactly what is converted, and what not: if the sentence “A device
working with 100 mW current” is converted into “A DEVICE WORKING
WITH 100 MW CURRENT”, this can have disastrous results (the M being
10^6 while m is 10^{-3}). Secondly one must know how to convert from lowercase
to uppercase, and vice-versa: for example, in Turkish, the uppercase version of
Latin ‘ı’ is ‘İ’ and not simply ‘I’ as in all other Latin languages. In Greek, accents
and breathings disappear when converted to uppercase, but hidden dieresis may
become apparent for reasons of disambiguation.
Once again, this is clearly a job for an ΩTP. But what about French text that has to be converted to uppercase? Obviously, ΩTPs have to be combinable, even better: nestable. This is precisely what Ω does: ΩTPs can be combined and nested so that in every context, however global or local, one can activate the appropriate one(s) to produce the appropriate results.

In the previous section, we mentioned encodings. These are not only a burden because of the multitude of font encodings, but also because text data which has to be typeset often is keyboarded in different encodings, according to location, language, OS, computer manufacturer, etc. Writing ΩTPs for converting between different encodings would certainly be possible, but in fact would be vain, since if there are \( n \) encodings, you would need \( 2n^2 \) different ΩTPs for all possible combinations—not to mention the problem of what to do with characters which are present in one encoding but not in the other...

There is a magnificent solution to this problem: Unicode. One of the design principles of Unicode is that it should contain all characters from all existing (data interchange) encodings. This means that whatever encoding the data is keyboarded in, one will always be able to convert it into Unicode, without losing any information. And of course the number of necessary ΩTPs is also strongly reduced: for \( n \) existing encodings, you need \( n \) ΩTPs, and for text already in the Unicode encoding, you need no conversion at all.

The various examples we have given show that there are three main categories of ΩTPs: those converting the data into Unicode (a process of “standardization” and “purification” of data), those transforming the data for linguistic or typographical reasons (this is best done once in Unicode encoding), and those converting the data from Unicode to the font encoding. The latter is obviously necessary since Unicode is not a font encoding (although there exist many fonts proclaiming themselves to be “Unicode-encoded”, which can only mean that they are not suitable for typography ...) but an information interchange encoding.

ΩTPs are part of Ω’s hidden machinery and are capable of solving most problems of (micro-)typography. Maybe the most important fact about them is that, while hidden to the end user, ΩTPs are entirely accessible and modifiable by the user. In fact, they are written in a very simple syntax, so that any user, without being a programmer, can write an ΩTP to solve a specific need of his/her. If for example, a Japanese Ω user has a long text with many furiganas to keyboard, he/she can invent a syntax for obtaining these by typing normal kana, marked up in some specific way. What way is not important, since the ΩTP he/she will write will convert it into \( \LaTeX \) commands; of course one could directly type \( \LaTeX \) commands, but (a) this would take longer, and (b) this would make the text unreadable. This method is efficient only because writing an ΩTP is a straightforward task.

But ΩTPs can also be quite complicated: for example the contextual analysis of Arabic, Syriac and Mongolian alphabets [that is the phenomenon of certain letters being connected and sometimes completely changing their forms] is also performed by ΩTPs. For most conventional software this task is accomplished by the operating system (this is the case of MacOS with Arabic Language Kit,
Arabic MacOS, Arabic Windows 95/98, or Windows 2000). Try to typeset an Arabic word under one of these operating systems and to color one of the letters: you will discover that contextual analysis breaks up, simply because the additional information inserted between letters cannot be discarded by the contextual analysis engine.

Modifying the contextual analysis engine of one of these operating systems can be difficult or even impossible, and is unlikely to be doable by the average end user. Performing contextual analysis using ΩTPs, even if the underlying operating system already provides a contextual analysis engine, gives the user, any user, total control over his text.

The story does not end here. A few years after ΩTPs were introduced and used massively, it became clear that, however powerful, their syntax was not sufficient for complex tasks. Here is a real-life example: as noted in the Introduction, in Thai language words are keyed in and typeset attached to each other, without any blank space or other visual delimiter. Nevertheless, Ω has to know their boundaries, so that when breaking lines and a word boundary is broken, a hyphen should be used. Detecting word boundaries in Thai is not a trivial task. Thai linguists have written software to do this, in the C language. Rewriting this software in ΩTP syntax would be a horrible waste of time.

This is why, very recently, a new kind of ΩTPs has been introduced: they are called “external” ΩTPs. They are used inside Ω exactly like any other ΩTP. The difference lies in the way they are written: in fact they are simply executables of the underlying operating system. Any executable can be used as a ΩTP, as long as it uses STDIN and STDOUT data streams. This means that the Thai executable, written in C, we just mentioned, can be used without any change, inside Ω. One does not even need to have the code to use such an executable: this means that in the future one may see “Ω plug-ins”: executables provided in binary form for selected platforms, enhancing the functionalities of Ω, and solving specific typesetting problems.

An example which is not specific to Thai and can be applied to any language: there is a public domain spell checker called ispell, and its international version ispell. The authors have written a small Perl script which uses ispell and, used as an ΩTP, colors words that are not recognized by ispell. This can be done without changing a given \TeX file, except for a single line at the beginning of the file, loading a given “macro-command package”. In this way, a user can prepare a \TeX file, run it with the “ispell plug-in”, see which words have been colored, correct eventual mistakes, and then run the file again without the plug-in, so that everything is in the appropriate color. This can be very useful because running a \TeX file through ispell can be very cumbersome: how can one distinguish text from \TeX commands, comments, mathematical formulas, etc.? There is a saying: “only \TeX can understand the \TeX programming language”, and this applies here: the only way to efficiently spellcheck a \TeX file is by doing it inside \TeX, or, in our case, Ω.

This short section has shown only a few examples of ΩTPs. Ω will keep the dual approach of internal, easy-to-write, platform-independent ΩTPs and external, arbitrarily complex and powerful, but platform-dependent ΩTPs.
the two of them provides the key to efficient multilingual typesetting in any context.

5 Generating MathML and XML

MathML is a W3C specification, available at http://www.w3c.org/Math/. It consists of two parts, the “presentation” part and the “content” part. For the moment, we will only consider the presentation part.

If all of \( \text{\LaTeX} \)'s mathematics used a prefix form for each of the different operators, then generating MathML would not be difficult. However, this is hardly the case: the most common examples are the infix operators \( \_\_\_ \) and \( \_\_\_\_\_ \) (and its variants). Furthermore, \( \text{\LaTeX} \) and \( \Omega \) consider that sub- and superscripts are placed on the last token in a stream, which does not at all correspond to the structure of the formula. Hence, \( \Omega \) must interpret formulas in order to generate useful MathML. Consider the example:

\[
(x_1+y_1)^2 - \gamma_0 \infty + \mathcal{F}_0^2
\]

giving the visible formula:

\[(x_1+y_1)^2 - \gamma_0 \infty + \mathcal{F}_0^2\]

The generated MathML is different:

\[
\langle mrow \rangle \langle msup \rangle \langle mrow \rangle \langle mo \rangle ( \langle /mo \rangle \langle mrow \rangle \langle msub \rangle \langle mi \rangle x \langle /mi \rangle \langle mn \rangle 1 \langle /mn \rangle \langle /msub \rangle \langle mo \rangle + \langle /mo \rangle \langle msub \rangle \langle mi \rangle y \langle /mi \rangle \langle mn \rangle 1 \langle /mn \rangle \langle /msub \rangle \langle mrow \rangle \langle mo \rangle ) \langle /mo \rangle \langle mrow \rangle \langle mn \rangle 2 \langle /mn \rangle \langle /msup \rangle \langle mo \rangle \langle /mo \rangle \langle msubsup \rangle \langle mi \rangle \&gamma; \langle /mi \rangle \langle mn \rangle 0 \langle /mn \rangle \langle /msubsup \rangle \langle mi \rangle \&infty; \langle /mi \rangle \langle msubsup \rangle \langle mo \rangle \langle /mo \rangle \langle msubsup \rangle \langle mi \rangle \&Scr\; \langle /mi \rangle \langle mn \rangle 0 \langle /mn \rangle \langle /msup \rangle \langle mi \rangle x \langle /mi \rangle \langle mn \rangle 2 \langle /mn \rangle \langle /msup \rangle \langle msubsup \rangle \langle /mrow \rangle
\]

As we can see, the

\[
(x_1+y_1)^2
\]

part was automatically transformed into:

\[
\{ \text{\texttt{left}}( \{ x_1+y_1 \} \text{\texttt{right}}) \}^{-2}
\]

To generate this text, no changes were made to \( \text{\LaTeX} \) or \( \text{\LaTeXe} \) macros. Everything was done by using the entities defined for each font glyph and by the changed typesetting engine.

\( \text{\LaTeXe} \) and \( \text{\LaTeX}\_\text{\LaTeXe} \) provide, in addition to the basic mathematics provided with \( \text{\LaTeX} \), all sorts of macros allowing one to define complex mathematical entities. When MathML is to be generated, one can redefine these macros so that they can directly generate the necessary structure. Consider, for example, the expression:
\[ \sqrt{\zeta+1} \]
\[ + \sqrt[3]{\zeta+1} \]

giving the visible form
\[ \sqrt{\zeta+1} + \sqrt[3]{\zeta+1} \]

The generated MathML is

\[
\text{<mrow> <msqrt> <mi>\zeta;</mi> <mo>+</mo> <mn>1</mn> </msqrt> \text{ </mrow>}
\text{ <mrow> <mroot> <mi>\zeta;</mi> <mo>+</mo> <mn>1</mn> </mroot> </mrow>}
\text{ <mrow> <mroot> <mi>\zeta;</mi> <mo>+</mo> <mn>3</mn> </mroot> </mrow>}
\]

This is done by redefining the \texttt{\textbackslash sqrt} macros so that it generates SGML start and end tags as appropriate.

In addition to being able to generate start and end tags for mathematics, it is also possible to generate start and end tags for horizontal material (normally used for plain text) as well as for vertical material (normally used for structuring). Using these tags, it is possible to translate a full \LaTeX\ document into HTML+MathML, XML+MathML, or any other sort of SGML-based markup. See the Web site for further details.