Draft documentation for the Ω system

John Plaice* Yannis Haralambous†

7 March 1998

1 Introduction

The Ω (Omega) typesetting system, an extension of Donald Knuth's TEX, is designed for the typesetting of all the world's languages. It normally uses the Unicode character encoding standard as internal representation, although it can accept any other character set for input or output. Since it allows one to dynamically define finite state automata to translate from one encoding to another, it is possible to define complex contextual analysis for ligature choice, character cluster building or diacritic placement, as required for scripts such as Arabic, Devanagari, Hebrew or Khmer. It also allows any number of transliterations, allowing anyone to type texts for any script, using any other script. Ω currently supports multidirectional writing, therefore allowing typesetting of Hebrew, Arabic, Chinese, Japanese, Mongolian and many other scripts.

A Unicode-based font is also being designed for the alphabetic scripts. This font is made up of four subfonts: (1) Latin, Greek, Cyrillic, Armenian, Georgian, punctuation; (2) Hebrew, Arabic, Syriac; (3) Dingbats and non-letterlike symbols; (4) Indic and South-East Asian scripts. This font consists of all the glyphs required to properly typeset each of the scripts, which means much more than designing one glyph for each Unicode position.

This document is the draft documentation for the Ω typesetting system, designed and developed by the authors. This draft document accompanies the 1.5 release of Ω, which is available at:

`ftp://ftp.cse.unsw.edu.au/users/plaice/Omega`

or at any of the CTAN sites.

This documentation should be considered cursory. In particular, it only describes the drivers that have been developed for typesetting and viewing, and only presents the tools that are based on `web2c`.

For more information, see our Web page, currently at:

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

*School of Computer Science and Engineering, University of New South Wales, Sydney 2052, Australia. plaice@cse.unsw.edu.au

†Atelier Fluxus Virus, 187, rue Nationale, F-59800 Lille, France. yannis@fluxus-virus.com

2 Implementation

The canonical Ω implementation is based on the standard `web2c` TEX distribution. Currently, Ω is based on `web2c-7.2`. This means that the following standard distributions automatically include Ω:

- Thomas Esser's TeTEX (Unix).
  Look up `http://www.tug.org/tetex/`
  or `mailto:te@informatik.uni-hannover.de` .
- Fabrice Popineau's TEXWin32 (Windows95/NT).
  Look up `ftp://ftp.ese-metz.fr/pub/TeX/win32`
  or `mailto:popineau@esemetz.ese-metz.fr` .
- Sebastian Rahtz's TEXLive (CD-ROM).
  Look up `http://www.tug.org/texlive.html`
  or `mailto:s.rahtz@elsevier.co.uk` .

In addition, there are currently two other prepackaged TEX environments that support Ω:

- Tom Kiffe's CMacΩ (MacIntosh).
  Look up `http://www.kiffe.com/cmacomega.html`
  or `mailto:tom@kiffe.com` .
- Christian Schenk's MiKTEX (Windows95/NT).
  Look up `http://www.inx.de/~cschenk/miktex`
  or `mailto:cschenk@snafu.de` .

The three files distributed with the Ω implementation are

```
web2c-7.2-omega-1.5.tar.gz
omegalib-1.5.tar.gz
omegadoc-1.5.tar.gz
```

To install Ω, you will require the standard TEX distribution as well. These files include

```
web-7.2.tar.gz
web2c-7.2.tar.gz
```

as well as a standard `texmf` tree. In addition to these files, the following drivers are needed:

```
dvipsk.tar.gz
odvipsk.tar.gz
gsftopk.tar.gz
xdvik.tar.gz
oxdvik.tar.gz
libwww.tar.gz
```

These files are all made available in the above `ftp` sites.

The installation procedure is described below. Assume that

- `/usr/local/ftp` contains your downloaded files;
- `/usr/local/src` is where you place source files; and
- `/usr/local/share` is where the `texmf` tree is to be placed;

```
FTP=/usr/local/ftp
SHARE=/usr/local/share
SRC=/usr/local/src
cd $SHARE
tar xzf $FTP/texmflib.tar.gz
tar xzf $FTP/omegalib-1.5.tar.gz
cd $SRC
tar xzf $FTP/web-7.2.tar.gz
tar xzf $FTP/web2c-7.2.tar.gz
tar xzf $FTP/web2c-7.2-omega-1.5.tar.gz
cd web2c-7.2
tar xzf $FTP/dvipsk.tar.gz
tar xzf $FTP/odvipsk.tar.gz
tar xzf $FTP/gsftopk.tar.gz
tar xzf $FTP/xdvik.tar.gz
tar xzf $FTP/oxdvik.tar.gz
tar xzf $FTP/libwww.tar.gz
configure
make
```

You will have to choose whether your call to `configure` needs any arguments.

3 What does Ω offer?

The Ω system is a derivative of Donald Knuth's TeX. As such, all of the TeX file types can be used by Ω as well. In addition there are six new file types. They are:

Suffix	Replaces	Description
`.opl`	`.pl`	Font property list (text)
`.ofm`	`.tfm`	Font metric (binary)
`.ovp`	`.vpl`	Virtual property list (text)
`.ovf`	`.vf`	Virtual font (binary)
`.otp`	——	Ω Translation Process (text)
`.ocp`	——	Ω Compiled Process (binary)

These different file types are described in future sections.

The Ω distribution contains several binaries, described below:

Binary	Replaces	Description
`omega` (Ω)	TeX	Typesetting engine (`.tex` → `.dvi`)
`lambda` (Λ)	LaTeX	For structured documents (`.tex` → `.dvi`)
`odvips`	`dvips`	PostScript driver (`.dvi` → `.ps`)
`oxdvi`	`xdvi`	Screen previewer for `.dvi` (`.dvi` → screen)
`odvicopy`	`dvicopy`	De-virtualizes `.dvi` (`.dvi` → `.dvi`)
`odvitype`	`dvitype`	Debugging for `.dvi` (`.dvi` → text)
`opl2ofm`	`pltotf`	Build font metric (`.opl` → `.ofm`)
`ofm2opl`	`tftopl`	Debugging for `.ofm` (`.ofm` → `.opl`)
`ovp2ovf`	`vptovf`	Build virtual font (`.ovp` → `.ofm` × `.ovf`)
`ovf2ovp`	`vftovp`	Debugging for `.ovf` (`.ofm` × `.ovf` → `.ovp`)
`otp2ocp`	——	Compile ΩTP (`.otp` → `.ocp`)
`outocp`	——	Debugging for `.ocp` (`.ocp` → text)
`mkofm`	`mktextfm`	Generate `.ofm` file if needed
`mkocp`	——	Generate `.ocp` file if needed

4 Sixteen-bit fonts, registers, etc.

One of the fundamental limitations of TeX3 is that most quantities can only range between 0 and 255. Fonts are limited to 256 characters each, only 256 fonts are allowed simultaneously, only 256 of any given kind of can be used simultaneously, etc. Ω loosens these restrictions, allowing 65 536 (0–65 535) of each of these entities.

4.1 Characters

Each font can allow up to 65 536 characters, ranging between 0 and 65 535. Unless other means are provided, using Ω Translation Processes (see section 8), the input and output mechanisms for characters between 256 (hex 100) and 65 535 (hex ffff) use four circumflexes. For example, `^^^^cab0` means hex value `cab0` and `^^^^0020` is the space character.

4.2 Fonts

Up to 65 536 fonts may be used. This is handled automatically, and space is allocated as needed.

4.3 Registers

Up to 65 536 registers of each kind may be used. The only case to be noted is that `\box255` remains the box used by the output routine.

4.4 Math codes

TeX allows the use of 16 (2^4) font families, each font of 256 (2^8) characters. To access the characters in the math fonts, and to define how they are to be used, there are several basic primitives:

- `\mathcode` ⟨*8-bit number*⟩ = ⟨*15-bit number*⟩:
  Defines 15-bit math code for character;
- `\mathcode` ⟨*8-bit number*⟩:
  Outputs 15-bit math code associated with character;
- `\mathchar` ⟨*15-bit number*⟩:
  Generates a math character with 15-bit math code;
- `\mathaccent` ⟨*15-bit number*⟩:
  Generates a math accent with 15-bit math code;
- `\mathchardef` ⟨*control-sequence*⟩ = ⟨*15-bit number*⟩:
  Defines a control sequence with a 15-bit math code;
- `\delcode` ⟨*8-bit number*⟩ = ⟨*27-bit number*⟩:
  Defines 27-bit delimiter code for character;
- `\delcode` ⟨*8-bit number*⟩:
  Outputs 27-bit delimiter code associated with character;
- `\delimiter` ⟨*27-bit number*⟩:
  Generates a math delimiter with 27-bit delimiter code;
- `\radical` ⟨*27-bit number*⟩:
  Generates a math radical with 27-bit delimiter code;

where

- ⟨*8-bit number*⟩ refers to an 8-bit character;
- ⟨*15-bit number*⟩ refers to value `0x8000` or a triple
  - 3 bits for math category,
  - 4 bits for font family,
  - 8 bits for character in font,

  called a *math code*;
- ⟨*27-bit number*⟩ refers to a negative number or a quintuple
  - 3 bits for math category,
  - 4 bits for first font family,
  - 8 bits for first character in font,

  - 4 bits for second font family,
  - 8 bits for second character in font,

  called a *delimiter code*.

Ω, on the other hand, allows 256 (2^8) font families, each font of 65 536 (2^{16}) characters. So, in addition to the TEX math font primitives, which continue to work, there are 16-bit versions:

- `\omathcode` ⟨*16-bit number*⟩ = ⟨*27-bit number*⟩:
  Defines 27-bit math code for character;
- `\omathcode` ⟨*16-bit number*⟩:
  Outputs 27-bit math code associated with character;
- `\omathchar` ⟨*27-bit number*⟩:
  Generates a math character with 27-bit math code;
- `\omathaccent` ⟨*27-bit number*⟩:
  Generates a math accent with 27-bit math code;
- `\omathchardef` ⟨*control-sequence*⟩ = ⟨*27-bit number*⟩:
  Defines a control sequence with a 27-bit math code;
- `\odelcode` ⟨*16-bit number*⟩ = ⟨*51-bit number*⟩:
  Defines 51-bit delimiter code for character;
- `\odelcode` ⟨*16-bit number*⟩:
  Outputs 51-bit delimiter code associated with character;
- `\odelimiter` ⟨*51-bit number*⟩:
  Generates a math delimiter with 51-bit delimiter code;
- `\oradical` ⟨*51-bit number*⟩:
  Generates a math radical with 51-bit delimiter code;

where

- ⟨*16-bit number*⟩ refers to a 16-bit character;
- ⟨*27-bit number*⟩ refers to value `0x8000000` or a triple
  - 3 bits for math category,
  - 8 bits for font family,
  - 16 bits for character in font,

  called a *math code*;
- ⟨*51-bit number*⟩ refers to a pair of numbers, either both negative or arranged as ⟨*27-bit number*⟩ ⟨*24-bit number*⟩, with the first number being:

- 3 bits for math category,
- 8 bits for first font family,
- 16 bits for first character in font,

and the second number being:

- 8 bits for second font family,
- 16 bits for second character in font,

called a *delimiter code*.

Since Ω is upwardly compatible with TeX, the older primitives still continue to function as expected. Internally, math codes are 27-bit numbers and delimiter codes are 51-bit numbers. However, if `\mathcode`⟨*15-bit number*⟩ appears in text mode, it continues to generate a 15-bit number, to remain upwardly compatible with TeX: Donald Knuth defines several numerical constants through `\mathcode`.

5 New typesetting routines

Most of the development in Ω has dealt with different means for manipulating character streams. Nevertheless, there are new typesetting routines.

5.1 New infinity level

A new infinity level `fi` has been added. It is smaller than `fil` but bigger than any finite quantity. Its original intention was for inter-letter stretching: either *filling-in-the-black*, as is done for calligraphic scripts such as Arabic; or for emphasis, as in Russian; all this without having to rewrite existing macro packages. There is therefore a new keyword, `fi`, and two new primitives, `\hfi` and `\vfi`.

5.2 Local paragraph parametrization

The Ω system allows the finetuning of layout, using *local* paragraph primitives. The first two, `\localinterlinepenalty` and `\localbrokenpenalty`, are generalizations of `\interlinepenalty` and `\brokenpenalty`.

When, say, `\localinterlinepenalty=200` appears, a *whatsit* node is deposited into the token list for the current paragraph. If the value is changed again, another whatsit node is deposited. When Ω cuts the paragraph into lines, it will add the current value of the local penalty to the penalty node that is placed after every line in the vertical list. Similarly for `\localbrokenpenalty` when a discretionary hyphen is placed at the end of a line. With these primitives, it becomes possible to discourage or encourage page breaks at more specific parts of a paragraph.

This same local approach is taken for a completely different task: placing fixed-width typeset material at the beginning (or the end) of every line in a paragraph.

« The original problem to be solved was for fine French typesetting, in which « guillemets are placed running down the left side of a paragraph, as in this « paragraph, so long as material is being quoted. » Since TeX breaks paragraphs in arbitrary places, it was impossible to develop a robust macro package that could, in a single pass, place the guillemets in the right positions.

The original text for the previous paragraph was:

```
{<<~\localleftbox{<<~}The original problem to be solved
was for fine French typesetting, in which guillemets
are placed running down the left side of a paragraph,
as in this paragraph, so long as material is being
quoted.~>>} Since \TeX\ breaks paragraphs in arbitrary
places, it was impossible to develop a robust macro
package that could, in a single pass, place the
guillemets in the right positions.
```

There are currently four local primitives:

- `\localleftbox{`⟨*typeset-material*⟩`}`:
  Until this primitive is redefined, then the typeset material will be placed at the beginning of every line that follows the occurrence of this primitive in the text.

- `\localrightbox{`⟨*typeset-material*⟩`}`:
  Until this primitive is redefined, then the typeset material will be placed at the end of every line that follows the occurrence of this primitive in the text.

- `\localinterlinepenalty` = ⟨*penalty*⟩:
  Until this primitive is redefined, then the given penalty value will be added to the penalty node placed between successive lines in a paragraph.

- `\localbrokenpenalty` = ⟨*penalty*⟩:
  Until this primitive is redefined, then each time that a line ends with a discretionary node, then the given penalty value will be added to the penalty node following that line.

Grouping is respected by all of the local paragraph primitives.

6 Multiple directions

Below is what is available in the experimental versions of Ω. Unfortunately we did not consider it to be sufficiently stable for it to be released generally. Therefore, Ω continues to support the bidirectionality functions of `TeX--XeT`. In addition, with the `\pagedirHR` and `\pagedirHL`, primitives, it is possible to transform the entire page into a right-to-left page or a left-to-right page. Similarly, `\pardirHR` and `\pardirHL` allow the paragraph direction to change. The page direction changes should occur in empty pages, and the paragraph direction changes should occur outside of horizontal mode. To ensure that tables are used properly, there is a primitive `nextfakemath`, which, put in front of math mode, ignores that the mathematics is supposed to be typeset from left-to-right. This is used in Λ, which goes into math mode to do `tabular` environments.

Since TEX was originally designed for English, it only supports left-to-right typesetting. This situation was improved somewhat with Knuth and MacKay's `TeX-XeT`*, modified into Breitenlohner's* `TeX--XeT`*. However, these modifications to TEX only allow the use of right-to-left typesetting, and even then, only within a particular paragraph. In other words, these systems do not support the typesetting of a full text in the different writing directions.*

The Ω system distinguishes sixteen different directions, which are designated by three parameters:

1. *The* beginning of the page *is one of T (top), L (left), R (right) or B (bottom). For English and Arabic, the beginning of the page is T; for Japanese it is R; for Mongolian it is L.*

2. *The* beginning of the line *defines where each line begins. For English, it is L; for Arabic, it is R; for Japanese and Mongolian, it is T.*

3. *The* top of the line *corresponds to the notion of 'up' within a line. Normally, this will be the same as for the beginning of the page, as in TLT for English, TRT for Arabic, RTR for Japanese, or LTL for Mongolian. However, for English included in Mongolian text, successive lines move 'up' the page, which gives direction LTR.*

The Ω system distinguishes three levels of different writing direction: page (`\pagedir`*), text (*`\textdir`*) and mathematics (*`\mathdir`*). Each of these primitives takes as primitive one of the above sixteen writing directions.*

- `\pagedir` ⟨direction⟩*: The page direction can only be changed if the current vlist is empty. This decision avoids ambiguous situations.*

- `\textdir` ⟨direction⟩*: This primitive can appear anywhere in a text, and Ω will allow for the moment only mixed horizontal combinations. Future versions will allow many different combinations, with parametrization. Grouping is respected, so it is possible to have inserts within a paragraph: these are implemented using the local paragraph mechanism described in the previous section.*

- `\mathdir` ⟨direction⟩*: Normally mathematics is done in the same direction as English, namely TLT. There have been situations where it has been written TRT. Ω allows only eight directions for mathematics, namely those in which the first and third direction parameters are identical.*

In addition, Ω allows one to designate the direction of a box. For example `\hbox dir TRT{...}` *creates a horizontal box, and uses direction* `TRT` *while building that box.*

Finally, fonts can be stored either naturally or not. In the unnatural situation, called with primitive `\unnaturaldir`*, it is understood that glyphs in the current font will always appear to the right of the current point, above the baseline. In the natural situation, called with* `\naturaldir`*, glyphs appear in the 'correct' direction. So a natural Arabic font would have the glyphs appear to the left of the current point, and a natural Japanese font would make the glyphs appear below the current point.*

7 Fonts for Ω

The TEX system takes the following approach to fonts. The TEX driver reads TEX documents and generates `.dvi` files. The driver uses font metric files (suffix `.tfm`, text version `.pl`) to determine how to lay out boxes on a pages. The screen driver or printer driver transforms the `.dvi` file in the appropriate format, using bitmap fonts (`.pk`), scaled fonts (`.pfa` or `.pfb`), or virtual fonts (`.vf`, text version `.vp`).

In the Ω system, we make no attempt, for the moment, to change the definition of bitmaps or scaled fonts. We have focused on the font metrics (`.ofm`, text version `.opl`), and the virtual fonts (`.ovf`, text version `.ovp`).

Currently, these new font file formats come in two versions. The first, called level 0, corresponds to the 16-bit version of TFM files, with no new functionality. Level 1 fonts are more ambitious, and provide for more powerful features, including compression methods and additional parameters.

7.1 Level-0 ΩFM files

The level-0 ΩFM files are simply 16-bit versions of TFM files, and have corresponding entries. Below is a description of the first 14 words of a level-0 ΩFM file. Each entry is a 32-bit integer, non-negative and less than 2^{31}:

$$
\begin{aligned}
\mathit{ofm\text{-}level} &= 0;\\
\mathit{lf} &= \text{length of the file, in words;}\\
\mathit{lh} &= \text{length of the header data, in words;}\\
\mathit{bc} &= \text{smallest character code in the font;}\\
\mathit{ec} &= \text{largest character code in the font;}\\
\mathit{nw} &= \text{number of entries in the width table;}\\
\mathit{nh} &= \text{number of entries in the height table;}\\
\mathit{nd} &= \text{number of entries in the depth table;}\\
\mathit{ni} &= \text{number of entries in the italic correction table;}\\
\mathit{nl} &= \text{number of entries in the lig-kern table;}
\end{aligned}
$$

nk	=	number of entries in the kern table;
ne	=	number of entries in the extensible character table;
np	=	number of font parameter words;
font-dir	=	direction of font.

We must have that $bc - 1 \leq ec \leq 65535$. Furthermore, the following identity must hold:

$$\begin{aligned} lf &= 14 + lh + 2 * (ec - bc + 1) + nw + nh + nd + ni + \\ &\quad 2 * nl + nk + 2 * ne + np. \end{aligned}$$

Note that a font may contain as many as 65536 characters (if $bc = 0$ and $ec = 65535$), and as few as 0 characters (if $bc = ec - 1$).

As with TFM files, if two or more octexts are combined to form an integer of 16 or more bits, the most significant octets appear first in the file. This is called BigEndian order.

Also as with TFM files, the rest of the file is a sequence of ten data arrays having the informal specification

header	:	**array** $[0..lh-1]$ **of** *stuff*
char-info	:	**array** $[bc..ec]$ **of** *char-info-word*
width	:	**array** $[0..nw-1]$ **of** *fix-word*
height	:	**array** $[0..nh-1]$ **of** *fix-word*
depth	:	**array** $[0..nd-1]$ **of** *fix-word*
italic	:	**array** $[0..ni-1]$ **of** *fix-word*
lig-kern	:	**array** $[0..nl-1]$ **of** *lig-kern-command*
kern	:	**array** $[0..nk-1]$ **of** *fix-word*
exten	:	**array** $[0..ne-1]$ **of** *extensible-recipe*
param	:	**array** $[1..np]$ **of** *fix-word*

There is no need to describe the entire file, only those parts that differ from TFM files: *char-info-word*, *lig-kern-command* and *extensible-recipe*. Here is a summary of those differences.

- *char-info-word* (8 octets):

width	16 bits
height	8 bits
depth	8 bits
italic	8 bits
RFU	6 bits
tag	2 bits
remainder	16 bits

  The meaning is as in TFM files, so there are 65536 possible widths, 256 possible widths, 256 possible heights and 256 possible italic corrections.

- *lig-kern-command* (8 octets):

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

  The meaning is as in TFM files, with every entry doubling in size.

- *extensible-recipe* (8 octets):

ext-top	16 bits
ext-mid	16 bits
ext-bot	16 bits
ext-rep	16 bits

  Once again, the meaning is as in TFM files, but every entry has been doubled.

7.2 Level-0 ΩPL files

The level-0 ΩPL files are the same as PL files, with the exception that values restricted to 8 bits can now be 16 bits.

7.3 Level-0 ΩVF files

The ΩVF files are indistinguishable from VF files, except for the file suffix. They exist only because the vast majority of drivers balk when they see characters that are not 8 bits.

7.4 Level-0 ΩVP files

The level-0 ΩVP files are the same as VP files, with the exception that values restricted to 8 bits can now be 16 bits.

7.5 Level-1 ΩFM files

The level-1 fonts take a different approach to level-0 fonts. They do not make the assumption that typesetting means simply placing placing glyphs on the baseline, one after another. Example applications include the automatic placement of glue between characters in East Asian scripts, the building of consonental clusters for South-Asian and South-East-Asian scripts, as well as the placing of diacritics in Arabic and Hebrew.

Level-1 fonts are different from level-0 fonts at three levels. First, they allow the definition of six new kinds of table:

- IVALUE tables contain integers.
- FVALUE tables contain fixword values that do not grow with magnification.
- MVALUE tables contain fixword values that do grow with magnification.

- RULE tables contain TEX rule definitions.
- GLUE tables contain TEX glue definitions.
- PENALTY tables contain TEX penalty definitions.

There can be several copies of each kind of table, but for the moment, there is a maximum of 32 new tables in all.

These new tables can be used as global tables, or can be indexed on a character-by-character basis in the *char-info-word* entries, which define character parameters. So, in addition to the standard parameters of width, height, depth and italic correction, additional parameters (of the six kinds outlined above) can be given for the characters.

To allow these new tables to be used, changes have also been made to the lig-kern table.

- Characters can be put into equivalence classes, where all characters in the same class will act the same in the lig-kern table;
- Glue nodes, rule nodes and penalty nodes can be inserted automatically into the stream, exactly as for kern nodes in TEX.
- The lig-kern program can be completely replaced by an ΩTP (see section 8).

Now we begin with the first part of the header of a level-1 ΩFM file. Here are the first 17 words of a level-1 ΩFM file. Each entry below is a 32-bit integer, non-negative and less than 2^{31}.

$$
\begin{aligned}
\textit{ofm-level} &= 1;\\
\textit{lf} &= \text{length of the file, in words;}\\
\textit{lh} &= \text{length of the header data, in words;}\\
\textit{bc} &= \text{smallest character code in the font;}\\
\textit{ec} &= \text{largest character code in the font;}\\
\textit{nw} &= \text{number of entries in the width table;}\\
\textit{nh} &= \text{number of entries in the height table;}\\
\textit{nd} &= \text{number of entries in the depth table;}\\
\textit{ni} &= \text{number of entries in the italic correction table;}\\
\textit{nl} &= \text{number of entries in the lig-kern table;}\\
\textit{nk} &= \text{number of entries in the kern table;}\\
\textit{ne} &= \text{number of entries in the extensible character table;}\\
\textit{np} &= \text{number of font parameter words;}\\
\textit{font-dir} &= \text{direction of font;}\\
\textit{nco} &= \text{offset of the character entries, in words;}\\
\textit{ncw} &= \text{number of character info words;}\\
\textit{npc} &= \text{number of parameters per character.}
\end{aligned}
$$

Most of the entries in the first part are as for level-0 fonts. The new entries pertain to how the *char-info-word* entries are stored.

- *nco*: This value gives the offset into the file for the first word of the *char-info-word* table. The *nco* value is required by output drivers, which need quick access to the characters, even if the total length of the tables preceding them is not easily computed,.
- *ncw*: Since many large fonts have large numbers of consecutive characters with identical metrics. These are compressed in level-1 fonts, and so the number of *char-info-word* entries is not simply $ec - bc + 1$. The *ncw* value gives the number of words used for character information, not the number of entries.
- *npc*: This is the number of extra parameters per character.
- *real-lf*: This would be the length of the file, were there no compression.

The next twelve entries come in pairs. For each kind of parameter (IVALUE, FVALUE, MVALUE, RULE, GLUE, PENALTY), the first entry states how many tables of that kind there are, and the second states how many words these tables require.

$$\begin{aligned}
nki &= \text{number of IVALUE tables;}\\
nwi &= \text{number of words for IVALUE tables;}\\
nkf &= \text{number of FVALUE tables;}\\
nwf &= \text{number of words for FVALUE tables;}\\
nkm &= \text{number of MVALUE tables;}\\
nwm &= \text{number of words for MVALUE tables;}\\
nkr &= \text{number of RULE tables;}\\
nwr &= \text{number of words for RULE tables;}\\
nkg &= \text{number of GLUE tables;}\\
nwg &= \text{number of words for GLUE tables;}\\
nkp &= \text{number of PENALTY tables;}\\
nwp &= \text{number of words for PENALTY tables.}
\end{aligned}$$

We must have that $bc - 1 \le ec \le 65535$. Furthermore, the following identity must hold:

$$\begin{aligned}
lf \quad = \quad & 29 + lh + ncw + nw + nh + nd + ni + \\
& 2 * nl + nk + 2 * ne + np + \\
& nki + nwi + nkf + nwf + nkm + nwm + \\
& nkr + nwr + nkg + nwg + nkp + nwp.
\end{aligned}$$

Finally, the sum $nki + nkf + nkm + nkr + nkg + nkp$ must be less than 32.

The rest of the file is composed of a number of arrays. The new parameter tables are placed before the standard dimension tables, as it is difficult to estimate space requirements without having read the new tables. Furthermore, the character parameter indices in the *char-info-word* entries are relative and must be translated into an absolute reference into the tables.

$$
\begin{array}{rcl}
\textit{header} & : & \textbf{array } [0..\textit{lh}-1] \textbf{ of } \textit{stuff} \\
\textit{ivalue-no} & : & \textbf{array } [0..\textit{nki}-1] \textbf{ of } \textit{integer} \\
\textit{fvalue-no} & : & \textbf{array } [0..\textit{nkf}-1] \textbf{ of } \textit{integer} \\
\textit{mvalue-no} & : & \textbf{array } [0..\textit{nkm}-1] \textbf{ of } \textit{integer} \\
\textit{rule-no} & : & \textbf{array } [0..\textit{nkr}-1] \textbf{ of } \textit{integer} \\
\textit{glue-no} & : & \textbf{array } [0..\textit{nkg}-1] \textbf{ of } \textit{integer} \\
\textit{pen-no} & : & \textbf{array } [0..\textit{nkp}-1] \textbf{ of } \textit{integer} \\
\textit{ivalue-table}[0] & : & \textbf{array } [0..\textit{ivalue-no}[0]-1] \textbf{ of } \textit{integer} \\
 & \vdots & \\
\textit{ivalue-table}[\textit{nki}-1] & : & \textbf{array } [0..\textit{ivalue-no}[\textit{nki}-1]-1] \textbf{ of } \textit{integer} \\
\textit{fvalue-table}[0] & : & \textbf{array } [0..\textit{fvalue-no}[0]-1] \textbf{ of } \textit{fix-word} \\
 & \vdots & \\
\textit{fvalue-table}[\textit{nkf}-1] & : & \textbf{array } [0..\textit{fvalue-no}[\textit{nkf}-1]-1] \textbf{ of } \textit{fix-word} \\
\textit{mvalue-table}[0] & : & \textbf{array } [0..\textit{mvalue-no}[0]-1] \textbf{ of } \textit{fix-word} \\
 & \vdots & \\
\textit{mvalue-table}[\textit{nkm}-1] & : & \textbf{array } [0..\textit{mvalue-no}[\textit{nkm}-1]-1] \textbf{ of } \textit{fix-word} \\
\textit{rule-table}[0] & : & \textbf{array } [0..\textit{rule-no}[0]-1] \textbf{ of } \textit{rule-entry} \\
 & \vdots & \\
\textit{rule-table}[\textit{nkr}-1] & : & \textbf{array } [0..\textit{rule-no}[\textit{nkr}-1]-1] \textbf{ of } \textit{rule-entry} \\
\textit{glue-table}[0] & : & \textbf{array } [0..\textit{glue-no}[0]-1] \textbf{ of } \textit{glue-entry} \\
 & \vdots & \\
\textit{glue-table}[\textit{nkg}-1] & : & \textbf{array } [0..\textit{glue-no}[\textit{nkg}-1]-1] \textbf{ of } \textit{glue-entry} \\
\textit{pen-table}[0] & : & \textbf{array } [0..\textit{pen-no}[0]-1] \textbf{ of } \textit{integer} \\
 & \vdots & \\
\textit{pen-table}[\textit{nkp}-1] & : & \textbf{array } [0..\textit{pen-no}[\textit{nkp}-1]-1] \textbf{ of } \textit{integer} \\
\textit{char-info} & : & \textbf{array } [0..\textit{ncw}-1] \textbf{ of } \textit{char-info-word} \\
\textit{width} & : & \textbf{array } [0..\textit{nw}-1] \textbf{ of } \textit{fix-word} \\
\textit{height} & : & \textbf{array } [0..\textit{nh}-1] \textbf{ of } \textit{fix-word} \\
\textit{depth} & : & \textbf{array } [0..\textit{nd}-1] \textbf{ of } \textit{fix-word}
\end{array}
$$

italic	:	**array** $[0..ni-1]$ **of** *fix-word*
lig-kern	:	**array** $[0..nl-1]$ **of** *lig-kern-command*
kern	:	**array** $[0..nk-1]$ **of** *fix-word*
exten	:	**array** $[0..ne-1]$ **of** *extensible-recipe*
param	:	**array** $[1..np]$ **of** *fix-word*

So, for parameter x, there is a table *x-no*, of length *nkx*, giving the size of each table. In addition, there are *nkx* tables containing the actual entries, where the i-th table is of length *x-no*$[i]$.

The only parameter entries with an unclear structure are *rule-entry* and *glue-entry*.

- Each *rule-entry* uses three words (12 octets):

1st word	*width*	32 bits	fixword
2nd word	*height*	32 bits	fixword
3rd word	*depth*	32 bits	fixword

  The interpretation of the values should be clear. If one of the three values is 0, then it can stretch in the appropriate direction, as is standard in TEX.

- Each *glue-entry* uses four words (16 octets):

1st word	*subtype*	4 bits	(0–3)
	argument-kind	4 bits	(0–2)
	stretch-order	4 bits	(0–4)
	shrink-order	4 bits	(0–4)
	char-rule	16 bits	
2nd word	*width*	32 bits	fixword
3rd word	*stretch*	32 bits	fixword
4th word	*shrink*	32 bits	fixword

  – *subtype* is one of

0	*normal*
1	*a-leaders*
2	*c-leaders*
3	*x-leaders*

  – *argument-kind* is one of

0	*space*
1	*rule* (*subtype* must be leader)
2	*character* (*subtype* must be leader)

– *stretch-order* and *shrink-order* are one of

0	*normal*
1	*fi*
2	*fil*
3	*fill*
4	*filll*

– n = *char-rule* depends on the value of *argument-kind*:

0. 0;
1. n-th rule in rule table 0;
2. n-character in font.

The explanation here only really makes sense if the reader has a clear understanding of how glue nodes are built in TEX. More detailed documentation is forthcoming.

The new *char-info-word* array is of great interest. Its length is not directly computable from the number of characters in the font. Each *char-info-word* entry contains a minimum of 12 octets, and is in any case a multiple of four octets. Each entry is as follows:

1st word	*width*	16 bits	
	height	8 bits	
	depth	8 bits	
2nd word	*italic*	8 bits	
	RFU	5 bits	
	ext-tag	1 bit	
	tag	2 bits	
	remainder	16 bits	
	no-repeats	16 bits	
	$param_0$	16 bits	
	...		
	$param_{npc-1}$	16 bits	
	padding	16 bits	if necessary

where *npc* is the number of characters per parameter.

The *repeat* entry allows one to state that the following `no-repeats` characters have identical attributes, thereby allowing the ΩFM file to be much smaller. This attribute is essential for Chinese, Japanese and korean ideogram fonts. In other words, this *char-info-word* entry is relevant to (*no-repeats* + 1) characters.

If the *ext-tag* bit is on, then the lig-kern entry pointed to by *remainder* is shared with all the other characters in its *equivalence class*, which corresponds to $param_0$ if there exists an IVALUE table.

We are now ready for the changed lig-kern table. There are four new instructions, which can be distinguished by the fact that the 0-th 16-bit entry (*skip-byte*) is exactly 256. In that case, then the 1st 16-bit entry (*next-char*) defines an equivalence class. If the next character is of that equivalence class, then the 2nd 16-bit entry (the *op-byte*) is interpreted as follows:

17. Add the glue node defined by entry *remainder* in the 0-th glue table.
18. Add the penalty node defined by entry *remainder* in the 0-th penalty table.
19. Add the penalty node defined by entry *remainder*/256 in the 0-th penalty table, then add the glue node defined by entry *remainder* mode 256 in the 0-th glue table.
20. Add the kern node defined by entry *remainder* in the 0-th mvalue table.

7.6 Level-1 ΩPL files

The level-1 ΩPL files are the text versions of level-1 ΩFM files. Hence, level-1 ΩPL files contain six kinds of new tables: integer (IVALUE), fixed (FVALUE), magnifiable fixed (MVALUE), rule (RULE), glue (GLUE) and PENALTY) tables. In addition, the character entries can include new parameters, which can then be used in the extended lig-kern table.

We begin with the new tables. These extra tables are numbered within each class, from 0 to $n-1$, where n is the number of tables in that class. To define, say, the fifth IVALUE table, one begins as follows:

(`FONTIVALUE H 5` ⟨*table-definition*⟩)

The instructions for defining tables are

(`FONTIVALUE` ⟨*table-no*⟩ ⟨*table-definition*⟩)
(`FONTFVALUE` ⟨*table-no*⟩ ⟨*table-definition*⟩)
(`FONTMVALUE` ⟨*table-no*⟩ ⟨*table-definition*⟩)
(`FONTRULE` ⟨*table-no*⟩ ⟨*table-definition*⟩)
(`FONTGLUE` ⟨*table-no*⟩ ⟨*table-definition*⟩)
(`FONTPENALTY` ⟨*table-no*⟩ ⟨*table-definition*⟩)

The property lists for these tables contain as many entries as there are slots in the table. So the fourth entry, starting from 0, in a glue table would begin as follows:

(`GLUE H 4` ⟨*glue-definition*⟩)

The instructions for defining entries are:

(`IVALUE` ⟨*entry-no*⟩ ⟨*ivalue-definition*⟩)
(`FVALUE` ⟨*entry-no*⟩ ⟨*fvalue-definition*⟩)
(`MVALUE` ⟨*entry-no*⟩ ⟨*mvalue-definition*⟩)
(`RULE` ⟨*entry-no*⟩ ⟨*rule-definition*⟩)
(`GLUE` ⟨*entry-no*⟩ ⟨*glue-definition*⟩)
(`PENALTY` ⟨*entry-no*⟩ ⟨*penalty-definition*⟩)

Now we come to the definitions of the individual entries. The four simple ones are for IVALUE, FVALUE, MVALUE and PENALTY, which are as follows: The instructions for defining entries are:

(`IVALUEVAL` ⟨*integer*⟩)
(`FVALUEVAL` ⟨*real*⟩)
(`MVALUEVAL` ⟨*real*⟩)
(`PENALTYVAL` ⟨*integer*⟩)

with some examples:

```
(IVALUEVAL H 42)
(PENALTYVAL D 1000)
(FVALUEVAL R 42.0)
(MVALUEVAL R 42.0)
```

which define an integer value of hex-42, a penalty value of 1000, a fix-word value of 42.0, and a magnifiable fix-word value of 42.0.

A ⟨*rule-definition*⟩ contains three components, each defaulting to 0:

(`RULEWD` ⟨*real*⟩)
(`RULEHT` ⟨*real*⟩)
(`RULEDP` ⟨*real*⟩)

The most complex entries are for glue, which can take several instructions. The first few instructions should be clear:

(`GLUEWD` ⟨*real*⟩)
(`GLUESTRETCH` ⟨*real*⟩)
(`GLUESHRINK` ⟨*real*⟩)
(`GLUESTRETCHORDER` ⟨*order*⟩)
(`GLUESHRINKORDER` ⟨*order*⟩)

where ⟨*order*⟩ is one of `UNIT`, `FI`, `FIL`, `FILL`, `FILLL`.

Now, glue can either be blank, or consist of a leader:

(`GLUETYPE` ⟨*kind*⟩)

where ⟨*kind*⟩ is one of `NORMAL`, `ALEADERS`, `CLEADERS`, `XLEADERS`. If a leader is chosen, then one of the following alternatives can be given:

(`GLUERULE` ⟨*integer*⟩)
(`GLUECHAR` ⟨*integer*⟩)

We give below the tables for an initial test with East Asian fonts:

```
(FONTIVALUE H 0
   (IVALUE H 0
      (IVALUEVAL H 0)
      )
   (IVALUE H 1
```

```
         (IVALUEVAL H 1)
         )
      (IVALUE H 2
         (IVALUEVAL H 2)
         )
      (IVALUE H 3
         (IVALUEVAL H 3)
         )
      )
   (FONTGLUE H 0
      (GLUE H 0
         (GLUETYPE H 0)
         (GLUESTRETCHORDER NORMAL)
         (GLUESHRINKORDER NORMAL)
         (GLUEWD R 0.0)
         (GLUESTRETCH R 0.0)
         (GLUESCHRINK R 0.0)
      )
      (GLUE H 1
         (GLUETYPE H 0)
         (GLUESTRETCHORDER NORMAL)
         (GLUESHRINKORDER NORMAL)
         (GLUEWD R 1.2333
         (GLUESTRETCH R 4.5555)
         (GLUESCHRINK R 2.3444)
      )
  (FONTPENALTY H 0
     (PENALTY H 0
         (PENALTYVAL H 0)
         )
     (PENALTY H 1
         (PENALTYVAL H 122A)
         )
     )
```

The extra tables can appear in any order, but they must all appear *before* the first character entry has appeared, since the character parameters can refer to these tables.

When defining the character entries, the standard entries (width, height, depth and italic correction) all exist. One can also add parameters to the characters by referring to the above tables. The syntax for an entry resembles

```
(CHARIVALUE H 0 H 2)
```

For this character, it is entry 2 in IVALUE table 0 that is relevant. All entries are similar:

$\big($`CHARIVALUE` $\langle integer\rangle$ $\langle integer\rangle\big)$
$\big($`CHARFVALUE` $\langle integer\rangle$ $\langle integer\rangle\big)$
$\big($`CHARMVALUE` $\langle integer\rangle$ $\langle integer\rangle\big)$
$\big($`CHARRULE` $\langle integer\rangle$ $\langle integer\rangle\big)$
$\big($`CHARGLUE` $\langle integer\rangle$ $\langle integer\rangle\big)$
$\big($`CHARPENALTY` $\langle integer\rangle$ $\langle integer\rangle\big)$

There is a special use for the 0-th integer table, which defines the equivalence class of the character for the lig-kern table:

$\big($`CHARIVALUE H 0` $\langle integer\rangle\big)$

The idea is that characters that act similarly with respect to their neighboring characters should have the same lig-kern entry, allowing for the dramatic reduction in size of the lig-kern table. More later.

Also to save space, it is possible to state that several characters use the same information. This is done with the CHARREPEAT instruction:

$\big($`CHARREPEAT H 34 H 42` $\langle$*character-definition*$\rangle\big)$

states that characters `0x34` through to `0x76` (`0x34`+`0x42`) all use the same information. This clustering is done automatically by the `ovp2ovf` program.

The lig-kern table uses four new instructions for the automatic insertion of kern, glue and penalties between characters. For example,

```
(CKRN H 3 H 2)
```

states that if we encounter this instruction, and the next character has 3 in its 0-th IVALUE table, then the 2-nd entry in the 0-th MVALUE table is inserted into the stream. Similarly,

```
(CGLUE H 3 H 2)
```

states that if we encounter this instruction, and the next character has 3 in its 0-th IVALUE table, then the 2-nd entry in the 0-th GLUE table is inserted into the stream. Once again,

```
(CPENALTY H 3 H 2)
```

does the same thing, except that it inserts the 2-nd entry in the 0-th PENALTY table into the stream. The other one is

```
(CPENGLUE H 3 H 2 H 4)
```

which inserts the 2-nd entry in the 0-th PENALTY table, then the 4-th entry in the 0-th GLUE table.

The LABEL instruction used in PL files has a variant called CLABEL, which means that several characters are using the same lig-kern entry. It is this technique that allows `ovp2ovf` to cluster the characters with similar properties, otherwise each would point to a different lig-kern entry.

Our example shows how East Asian fonts might be coded. The equivalence class of a character has three possible values: 1 for 'left' characters (opening parenthesis, opening quote, etc.), 2 for 'middle' or ordinary characters, and 3 for 'right' characters (closing parenthesis, closing quote, period, etc.). Here is the lig-kern table.

```
(LIGTABLE
   (CLABEL H 1)
   (CPENGLUE H 1 H 0 H 0)
   (CPENGLUE H 2 H 0 H 0)
   (CPENGLUE H 3 H 0 H 0)
   (STOP)
   (CLABEL H 2)
   (CGLUE H 1 H 0)
   (CGLUE H 2 H 0)
   (CPENGLUE H 3 H 0 H 0)
   (STOP)
   (CLABEL H 2)
   (CGLUE H 1 H 0)
   (CGLUE H 2 H 0)
   (CPENGLUE H 3 H 0 H 0)
   (STOP)
```

Glue is inserted between all pairs of characters that are of category 1, 2, or 3. In addition, a penalty is added in front of characters of category 3 ('right' characters), preventing a linebreak just prior to such characters. At the same time, a penalty is added after all occurrences of characters of category 1 ('left' characters).

Another possibility is to completely replace the lig-kern table, with the instruction

$\big($`LIGTABLEOCP` $\langle$*ocp-file-name*$\rangle\big)$

Here the ΩCP $\langle$*ocp-file-name*$\rangle$ will be used instead of the lig-kern table.

7.7 Level-1 ΩVF files

The level-1 ΩVF files are indistinguishable from level-0 ΩVF files.

7.8 Level-1 ΩVP files

The level-1 ΩVP files are similar to level-1 ΩPL files for the description of the tables. For the actual character layout stuff, there is no difference with level-0 ΩVP files.

8 Ω Translation Processes

The changes described above are very useful, and allow the resolution of several problems. However, they do not radically alter the structure of TEX. This is not the case for the Ω Translation Processes, which allow text to be passed through any number of finite state automata, in order to impose the required effects.

These processes are necessary for translating one character set to another. They are also used to choose the various forms of letters in Arabic, or to create consonental clusters in Khmer, or to rearrange letter order in Indic scripts. They could also offer alternative means of changing texts to upper or lower case or to hyphenate texts.

Each translation process is placed in a file with the suffix `.otp`. Its syntax is similar but not identical to a `lex` or `flex` file on Unix. Examples of translation processes can be found in the `texmf/omega/otp` directory.

An `.otp` file defines a finite state automaton that transforms an input character stream into an output character stream. It consists of six parts:

Input
Output
Tables
States
Aliases
Expressions

where the *Expressions* actually state what translations take place and in what situation.

In what follows, n refers to a positive integer between 0 and $2^{24} - 1$. It can be given in decimal form, octal form (preceded by `@'`) or hexadecimal form (preceded by `@"`). Hexadecimal numbers can use both minuscule and majuscule letters to express the digits *a–f*. Numbers can also be given in character form: a printable ASCII character, when placed inside a pair of quotes, generates the ASCII code for that character. For example, `'a'` is equivalent to `@"61`.

The *Input* part states how many octets are in each input character. If the section is empty, then the default value is 2, since we hope that Unicode will become the standard means of communication in the future. If the section is not empty, it must be of the form

`input:` *in*`;`

where *in* states how many octets are in each input character.

The *Output* part states how many octets are in each output character. If the section is empty, then the default value is 2, since we hope that Unicode will become the standard means of communication in the future. If the section is not empty, it must be of the form

`output:` *out*`;`

where *out* states how many octets are in each output character.

The *Tables* part is used for defining tables that will be referred to later in the expressions. Often, translations from one character set to another are most efficiently presented through table lookup. This section can be empty, in which case no tables have been defined. If it is not empty, it is of the form

`tables:` $table^+$

where each *table* is of the form

id`[`n`] = {`n^+`};`

where the numbers in n^+ are comma-separated.

The *States* part is used to separate out the expressions. Not all expressions will necessarily be applicable in all situations. To do this, the user can name states and identify expressions with state names, in order to express what expressions apply when. This section can be empty, in which case there is only one state. If it is not empty, it is of the form

`states:` id^+`;`

where the identifiers in id^+ are comma-separated.

The *Aliases* part is used to simplify the definition of the left hand sides of the expressions. Each expression consists of a left-hand side, in the form of a simplified regular expression, and of a right-hand side, which states what should be done with a recognized string. To simplify the definitions of the left-hand sides, aliases can be used. This section can be empty, in which case there are no aliases. If it is not empty, it is of the form

`aliases:` $alias^+$

where each *alias* is of the form

id `=` *left*`;`

and *left* is defined below.

The *Expressions* part is the very reason for an `.otp` file. It states what translations must take place, and when. It cannot be empty, and its syntax is

`expressions:` $expr^+$

Each *expr* is of the form

leftState totalLeft right pushBack rightState`;`

where *leftState* defines the state for which this expression is applicable, *totalLeft* defines the left-hand-side regular expression, *right* defines the characters to be output, *pushBack* states what characters must be added to the input stream and *rightState* gives the new state.

Intuitively, if the automaton is in macro-state *leftState* and the regular expression *totalLeft* corresponds to a prefix of the current input stream, then (1) the input stream is advanced to the end of the recognized prefix, (2) the characters generated by the *right* expression are put onto the output stream,

(3) the characters generated by the *pushBack* stream are placed at the beginning of the input stream and (4) the system changes to the macro-state defined by *rightState*.

The *leftState* field can be empty. If it is not, its syntax is

<*id*>

The syntax for *totalLeft* is

`beg:`? *left*$^+$ `end:`?

The `beg:`, if present, will only match the string if it is at the beginning of the input. The `end:`, if present, will only match the string if it is at the end of the input.

The syntax for *left* is given by

$$
\begin{array}{lcl}
\mathit{left} & ::= & n \\
 & | & n\texttt{-}n \\
 & | & \texttt{.} \\
 & | & \texttt{(}\mathit{left}^{+}\texttt{)} \\
 & | & \texttt{\^{}(}\mathit{left}^{+}\texttt{)} \\
 & | & \{\mathit{id}\} \\
 & | & \mathit{left}\ \texttt{<}n\texttt{,}n?\texttt{>}
\end{array}
$$

where the *left*$^+$ means a series of *left* separated by vertical bars. Therefore, n means a single number, n-n is a range, `.` is a wildcard character, `(`*left*$^+$`)` is a choice, `^(`*left*$^+$`)` is the negation of a choice, `{`*id*`}` is the use of an alias and *left*<n,n?> means between n and n' occurrences of *left*. Should there be no n', then the expression means at least n occurrences.

The syntax for *right* is

=> *stringExpr*$^+$

while that for *pushBack*, if it is not empty, is

<= *stringExpr*$^+$

The *right* expression corresponds to the characters that are to be output. The *pushBack* expression corresponds to the characters that are put back onto the input stream.

A *stringExpr* defines a string of characters, using the characters in the recognized input stream as arguments. It is of the form

s
$\mid$ n
$\mid$ `\`n
$\mid$ `\$`
$\mid$ `\($-`n`)`
$\mid$ `\*`
$\mid$ `\(*-`n`)`
$\mid$ `\(*+`n`)`
$\mid$ `\(*+`n`-`n'`)`
$\mid$ `#`*arithExpr*

where s is an ASCII character string enclosed in double quotation marks. The `\`n means the n-th character (starting from 1) in the recognized prefix; the `\$` means the last character in the prefix; `\($-`n`)` the n-th, counting from the end. The `\*` means the entire recognized prefix; `\(*-`n`)` the prefix without the last n characters; `\(*+`n`)` without the first n characters; `\(*+`n`-`n'`)` removes the first n and last n' characters.

For example, Indic scripts are encoded with vowels at the end of a syllable, but the vowel is actually printed first on the page. Up to six consonants can precede a vowel, yielding the following transliteration:

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

The *arithExpr* entry allows for calculations to actually be effected on the characters in the prefix. Their syntax is as follows:

n
$\mid$ `\`n
$\mid$ `\$`
$\mid$ `\($-`n`)`
$\mid$ *arithExpr* `+` *arithExpr*
$\mid$ *arithExpr* `-` *arithExpr*
$\mid$ *arithExpr* `*` *arithExpr*
$\mid$ *arithExpr* `div:` *arithExpr*
$\mid$ *arithExpr* `mod:` *arithExpr*
$\mid$ *id*`[`*arithExpr*`]`
$\mid$ `(`*arithExpr*`)`

where *id*`[`*arithExpr*`]` means a table lookup: the *id* must be a table defined in the *Tables* section. The other operations should be clear.

The following example shows the use of tables.

```
% File inbig5.otp
% Conversion to Unicode from Chinese Big 5 (HKU)
% Copyright (c) 1995 John Plaice and Yannis Haralambous
% This file is part of the Omega project.
%
% This file was derived from data in the tcs program
% ftp://plan9.att.com/plan9/unixsrc/tcs.shar.Z, 16 November 1994
%
```

```
input:  1;
output: 2;

tables:

in_big5_a1[@"9d] = {
@"20,   @"2c,   @"2ce,  @"2e,   @"2219, @"2219, @"3b,   @"3a,
...
@"2199, @"2198, @"2225, @"2223, @"2215
};

in_big5[@"3695] = {
@"3000, @"ff0c, @"3001, @"3002, @"ff0e, @"30fb, @"ff1b, @"ff1a,
...
@"fffd, @"fffd, @"fffd, @"fffd, @"fffd
};

expressions:

@"1a                     => @"0a;
@"00-@"a0                => \1;
@"a1(@"40-@"7e)          => #(in_big5_a1[\2-@"40]);
@"a1(@"a1-@"fe)          => #(in_big5_a1[\2-@"62]);
(@"a2-@"fe)(@"40-@"7e)  => #(in_big5[(\1-@"a2)*@"9d + \2-@"40]);
(@"a2-@"fe)(@"a1-@"fe)  => #(in_big5[(\1-@"a2)*@"9d + \2-@"62]);
. .                      => @"fffd;
```

In the future, more operations may well be added. Research is still under way for such things as providing means for defining functions, local variables, error handling and other functionality.

The *pushBack* part, which serves to put characters back onto the input stream, uses the same syntax as the *right* part. When characters are placed back onto the input stream, they will be looked at upon the next iteration of the automaton.

Finally, the *rightState* can be empty or one of the following three forms:

<*id*>
| `<push:` *id*`>`
| `<pop:>`

If it is empty, the automaton stays in the same state. If it is of the form <*id*>, then the automaton changes to state *id*. The `<push:` *id*`>` means change to state *id*, but remembering the current state. The `<pop:>` means return to the previously saved state.

Several `.otp` files are in the `omega/texmf/otp` directory. The `char2uni` directory contains ΩTPs that convert national character sets to Unicode, while the `omega` directory contains ΩTPs designed to work with the Ω fonts.

9 Compiled Translation Processes.

Ω does not know anything about Ω Translation Processes. It actually reads a compiled form of these filters, known as Compiled Translation Processes (file suffix `.ocp`). Essentially, the ΩCPs can be considered to be portable assembler programs, and Ω includes an interpreter for the generated instructions.

The command for reading in a ΩCP file is similar to a font declaration. The example

```
\ocp\TexUni=TeXArabicToUnicode
```

means that the file `TeXArabicToUnicode.ocp` is read in by Ω and that internally the translation process is referred to as `\TeXUni`.

The ΩCPs consist of a sequence of 4-octet words. The first seven words have the following form:

lf	length of the entire file, in words;
in	number of octets in an input character;
ot	number of octets in an output character;
nt	number of tables;
lt	number of words allocated for tables;
ns	number of states;
ls	number of words allocated for states;

The header words are followed by four arrays:

$$\begin{aligned}
\mathit{table_length} &: \textbf{array } [0..nt-1] \textbf{ of } \mathit{word}\\
\mathit{tables} &: \textbf{array } [0..lt-1] \textbf{ of } \mathit{word}\\
\mathit{state_length} &: \textbf{array } [0..ns-1] \textbf{ of } \mathit{word}\\
\mathit{tables} &: \textbf{array } [0..ls-1] \textbf{ of } \mathit{word}
\end{aligned}$$

The *table_length* array states how many words are used for each of the tables in the ΩCP. For the GB $\to$ Unicode example on page 26, the *table_length* would have two entries: hex values `9d` and `3695`.

The *tables* array is simply the concatenation of the tables in the ΩTP file.

The *state_length* array states how many words are used for each of the states in the ΩCP. For the GB $\to$ Unicode example on page 26, the *state_length* would have one entry.

The *states* array is simply the concatenation of the sequence of instructions for each state in the ΩTP file. Each instruction takes one or two 4-octet words. Zero- and one-argument instructions use one word. If the instruction consists of one word, then the actual instruction is in the first two octets and the argument is in the last two octets. If the instruction consists of two words, then the actual instruction is in the first two octets, the first argument is in the next two octets and the last argument is in the last two octets. The instructions are as follows:

1	`OTP_RIGHT_OUTPUT`	0 arguments
2	`OTP_RIGHT_NUM`	1 argument
3	`OTP_RIGHT_CHAR`	1 argument

4	OTP_RIGHT_LCHAR	1 argument
5	OTP_RIGHT_SOME	2 arguments
6	OTP_PBACK_OUTPUT	0 arguments
7	OTP_PBACK_NUM	1 argument
8	OTP_PBACK_CHAR	1 argument
9	OTP_PBACK_LCHAR	1 argument
10	OTP_PBACK_SOME	2 arguments
11	OTP_ADD	0 arguments
12	OTP_SUB	0 arguments
13	OTP_MULT	0 arguments
14	OTP_DIV	0 arguments
15	OTP_MOD	0 arguments
16	OTP_LOOKUP	0 arguments
17	OTP_PUSH_NUM	1 argument
18	OTP_PUSH_CHAR	1 argument
19	OTP_PUSH_LCHAR	1 argument
20	OTP_STATE_CHANGE	1 argument
21	OTP_STATE_PUSH	1 argument
22	OTP_STATE_POP	1 argument
23	OTP_LEFT_START	0 arguments
24	OTP_LEFT_RETURN	0 arguments
25	OTP_LEFT_BACKUP	0 arguments
26	OTP_GOTO	1 argument
27	OTP_GOTO_NE	2 arguments
28	OTP_GOTO_EQ	2 arguments
29	OTP_GOTO_LT	2 arguments
30	OTP_GOTO_LE	2 arguments
31	OTP_GOTO_GT	2 arguments
32	OTP_GOTO_GE	2 arguments
33	OTP_GOTO_NO_ADVANCE	1 argument
34	OTP_GOTO_BEG	1 argument
35	OTP_GOTO_END	1 argument
36	OTP_STOP	0 arguments

The `OTP_LEFT`, `OTP_GOTO` and `OTP_STOP` instructions are used for recognizing prefixes in an input stream. The `OTP_RIGHT` instructions place characters on the output stream, while the `OTP_PBACK` instructions place characters back onto the input stream. The instructions `OTP_ADD` through to `OTP_PUSH_LCHAR` are used for internal computations in preparation for `OTP_RIGHT` or `OTP_PBACK` instructions. Finally, the `OTP_STATE` instructions are for changing macro-states.

The system that reads from the input stream uses two pointers, which we will call *first* and *last*. The *first* value points to the beginning of the input prefix that is currently being identified. The *last* value points to the end of the input prefix that has been read. When a prefix has been recognized, then *first* points to `\1` and *last* points to `\$`.

The `OTP_LEFT_START` instruction, called at the beginning of the parsing of a prefix, advances *first* to $last + 1$; `OTP_LEFT_RETURN` resets the *last* value to $first - 1$ (it is called when a particular *left* pattern does not correspond to the prefix); `OTP_LEFT_BACKUP` backs up the *last* pointer by 1.

Internally, a ΩCP program uses a program counter (PC), which is simply an index into the appropriate state array. Like for all assembler programs, this counter is normally incremented by 1 or 2, depending on the size of the instruction, but it can be abruptly changed through an `OTP_GOTO` instruction.

The argument in single-argument `OTP_GOTO` instructions is the new PC. For the two-argument instructions, the first is the comparand and the second is the new PC should the test succeed. The `OTP_GOTO` instruction itself is an unconditional branch; `OTP_GOTO_NO_ADVANCE` advances *last* by 1, and branches if has reached the end of input; `OTP_GOTO_BEG` branches at the beginning of input and `OTP_GOTO_END` branches at the end of input. As for `OTP_GOTO_`*cond*, it succeeds if the character pointed to by *last* (we'll call it $*last$) satisfies the test $cond(*last, firstArg)$.

The `OTP_STOP` instruction stops processing of the currently recognized prefix. Normally the automaton will be restarted with an `OTP_LEFT_START` instruction.

When computations are undertaken for the `OTP_RIGHT` and `OTP_PBACK` instructions, a computation stack is used. This stack is accessed through instructions `OTP_ADD` through to `OTP_PUSH_LCHAR`, as well as through the instructions `OTP_RIGHT_OUTPUT` and `OTP_PBACK_OUTPUT`.

Since the `OTP_RIGHT` and `OTP_PBACK` instructions are analogous, only the former are described. The `OTP_RIGHT_OUTPUT` instruction pops a value of the top of the stack and outputs it; `OTP_RIGHT_NUM`(n) simply places n on the output stream; `OTP_RIGHT_CHAR`(n) places the n-th input character on the output stream; `OTP_RIGHT_LCHAR` does the same, but from the back; finally, `OTP_RIGHT_SOME` places a substring onto the output stream.

Three instructions are used for placing values on the stack: `OTP_PUSH_NUM`(n) pushes n onto the stack, `OTP_PUSH_CHAR`(n) pushes the n-th character and `OTP_PUSH_LCHAR`(n) does the same from the end.

The arithmetic operations of the form `OTP_`*op* apply the operation

$$stack[top-1] \quad := \quad stack[top-1] \; op \; stack[top]$$

where *top* is the stack pointer, and then decrement the stack pointer. Finally, the `OTP_LOOKUP` instruction applies the operation

$$stack[top-1] \quad := \quad stack[top-1][stack[top]]$$

and then decrements the pointer.

Last, but not least, are the `OTP_STATE` instructions, which manipulate a stack of macro-states. The initial state is always 0. The `OTP_STATE_CHANGE`(n)

changes the current state state n; `OTP_STATE_PUSH`(n) pushes the current state onto the state stack before changing the current state; `OTP_STATE_POP` pops the state at the top of the state stack into the current state.

10 Translation process lists

Translation processes can be used for a number of different purposes. Since not all uses can be foreseen, we have decided to offer a means to dynamically reconfigure the set of translation processes that are passing over the input text. This is done using stacks of translation process lists.

For any single purpose, for example to process a given language, several ΩCPs might be required. If one makes a context switch, such as processing a different language, then one would to be able to quickly replace *all* of the ΩCPs that are currently being used. This is done using ΩCP lists.

A ΩCP list is actually a list of pairs. Each pair consists of a positive scaled value and a doubly ended queue of ΩCPs. For example,

```
\ocplist\ArabicOCP=[(1.0 : \TexUni,\UniUniTwo,\UniTwoFont)]
```

the output from Ω once the ΩCP list `\ArabicOCP` has been typed, shows that that list has one element, namely the pair with the scaled value 1.0 and the doubly ended queue with three ΩCPs, `\TexUni`, `\UniUniTwo` and `\UniTwoFont`.

ΩCP lists are built up using the five operators `\nullctlist`, `\addbefore-ocplist`, `\addafterocplist`, `\removebeforeocplist` and `\removeafterocp-list`. For example, the above output was generated by the following sequence of Ω statements:

```
\ocp\TexUni=TeXArabicToUnicode
\ocp\UniUniTwo=UnicodeToContUnicode
\ocp\UniTwoFont=ContUnicodeToTeXArabicOut

\ocplist\ArabicOCP=
\addbeforeocplist 1 \TexUni
\addbeforeocplist 1 \UniUniTwo
\addbeforeocplist 1 \UniTwoFont
\nullocplist
```

The `\ocplist` command is similar to the `\ocp` command:
`\ocplist` *listName* = *ocpListExpr*.

All *ocpListExpr* are built up from either the empty ΩCP list, `\nullocplist`, or from an already existing ΩCP list. In the latter case, the list is completely copied, to ensure that the named list is not itself modified. Given a list l, the instruction `\addbeforeocplist` n *ocp* l states that the ΩCP *ocp* is added at the head of the doubly ended queue for value n in list l. If that queue does not exist, it is created and inserted in the list so that the scaled values are all in increasing order. The instruction `\addafterocplist` n *ocp* l does the same, except the addition takes place at the tail of the doubly ended queue. The instruction

`\removebeforeocplist` n l removes the ΩCP at the head of the doubly ended queue numbered n. The instruction `\removeafterocplist` n l does the same at the tail of the doubly ended queue. See the next section for more examples.

11 Input Filters

Here we come to the crucial parts of Ω. What happens to the input stream as it passes through translation processes? What is the interaction between TEX's macro-expansion and Ω's translation processes?

When Ω is in horizontal mode and it encounters a token of the form *letter*, *other_char*, *char_given* or *char_num*, that character and all the successive characters in those categories are read into a buffer. The currently active ΩCP is applied to the buffer, and the result is placed back onto the input, to be reread by the standard TEX input routines, including macro expansion.

The currently active ΩCP is designated by a pair (v, i), where v is a scaled value and i is an integer. If all the enabled ΩCPs are in a ΩCP list, then the v designates the index into the ΩCP list and the i designates which element in the v-queue is currently active.

Once a ΩCP has been used, the i is incremented; if it points to the end of the current queue, then v is set to the next queue, and i is reset to 1.

When the last enabled ΩCP has been used, then the standard techniques for treating letters and other characters are used, namely generating paragraphs, etc.

What this means is that it is now possible to apply a filter on the *text* of a file without macro-expansion, generate a new text, possibly with macros to be expanded, macro-expand, re-apply filters, etc. All this without active characters, and without breaking macro packages.

How are ΩCP lists enabled? ΩCP lists are placed on a stack, each numbered queue in a given list masking the queues with the same number for the lists below that one on the stack.

There are three commands, which all respect the grouping mechanism. The `\clearocplists` command disables all ΩCP lists. The `\pushocplist` *OCPlist* command pushes *OCPlist* onto the stack. The `\popocplist` command pops the last list from the stack.

For example, consider the following purely hypothetical situations:

```
\ocplist\FrenchOCP = \addbeforeocplist 1 \ocpA
                     \addbeforeocplist 2 \ocpB
                     \addbeforeocplist 3 \ocpC
                     \nullocplist

\ocplist\GermanOCP = \addbeforeocplist 1 \ocpD
                     \addbeforeocplist 2 \ocpE
                     \addbeforeocplist 3 \ocpF
                     \nullocplist
```

```
\ocplist\ArabicOCP = \addbeforeocplist 1 \ocpG
                     \addbeforeocplist 2 \ocpH
                     \addbeforeocplist 2 \ocpI
                     \addbeforeocplist 3 \ocpJ
                     \nullocplist

\ocplist\SpecialArabicOCP =
                     \addafterocplist 3 \ocpK
                     \ArabicOCP

\ocplist\UpperCaseOCP =
                     \addbeforeocplist 2.5 \ocpL
                     \nullocplist
```

There are now 5 ΩCP lists *defined*, but none of them are *enabled*. The defined lists are:

```
\ocplist\FrenchOCP =
    [(1.0:\ocpA), (2.0:\ocpB), (3.0:\ocpC)]
\ocplist\GermanOCP =
    [(1.0:\ocpD), (2.0:\ocpE), (3.0:\ocpF)]
\ocplist\ArabicOCP =
    [(1.0:\ocpG), (2.0:\ocpH,\ocpI), (3.0:\ocpJ)]
\ocplist\SpecialArabicOCP =
    [(1.0:\ocpG), (2.0:\ocpH,\ocpI), (3.0:\ocpJ,\ocpK)]
\ocplist\UpperCaseOCP =
    [(2.5:\ocpL)]
```

Consider now the sequence of instructions

```
\clearocplists
\pushocplist\FrenchOCP
\pushocplist\UpperCaseOCP
\pushocplist\GermanOCP
\popocplist
\popocplist
\pushocplist\ArabicOCP
\pushocplist\SpecialArabicOCP
\pushocplist\GermanOCP
```

The effective enabled ΩCP list is, in turn:

```
[]
[(1.0:\ocpA), (2.0:\ocpB), (3.0:\ocpC)]
[(1.0:\ocpA), (2.0:\ocpB), (2.5:\ocpL), (3.0:\ocpC)]
[(1.0:\ocpD), (2.0:\ocpE), (2.5:\ocpL), (3.0:\ocpF)]
[(1.0:\ocpA), (2.0:\ocpB), (2.5:\ocpL), (3.0:\ocpC)]
[(1.0:\ocpA), (2.0:\ocpB), (3.0:\ocpC)]
```

```
[(1.0:\ocpG), (2.0:\ocpH,\ocpI), (3.0:\ocpJ)]
[(1.0:\ocpG), (2.0:\ocpH,\ocpI), (3.0:\ocpJ,\ocpK)]
[(1.0:\ocpD), (2.0:\ocpE), (3.0:\ocpF)]
```

The first test of the ΩCP lists was for Arabic. The text was typed in ASCII, using a Latin transliteration. This text was first transformed into Unicode, the official 16-bit encoding for the world's character sets. These letters were then translated into their appropriate visual forms (isolated, initial, medial or final) and then the text was translated into the font encoding. During the second translation, inter-letter black spacing is inserted, since Arabic typesetting calls for word expansion to fill out a line. Here is the input:

```
\font\ARfont=oar10 scaled 1728 offset 256 %% an X-font
\def\keshideh{%
\begingroup\penalty10000%
\clearocplists\xleaders\hbox{\char'767}\hskip0ptplus1fi%
\endgroup}
\ocp\TexUni=TeXArabicToUnicode
\ocp\UniUniTwo=UnicodeToContUnicode
\ocp\UniTwoFont=ContUnicodeToTeXArabicOut
\ocplist\ArabicOCP=%
\addbeforeocplist 1 \TexUni
\addbeforeocplist 1 \UniUniTwo
\addbeforeocplist 1 \UniTwoFont
\nullocplist
\def\AR#1{\begingroup\noindent\pushocplist \ArabicOCP%
\ARfont\language=255\textdir TRT #1\endgroup}
```

Notice that the `\keshideh`, which is dynamically inserted between letters by the `\UniUniTwo` ΩCP, uses the `fi` infinity. It also disables all of the ΩCPs, within a group.

12 Automatic detection of character sets

Most character sets belong to one of three groups:

1. 8-bit character sets (including shift character sets) that include ASCII;
2. 8-bit character sets (including shift character sets) that include EBCDIC; and
3. 16-bit character sets that include ASCII as the first 128 characters, such as Unicode. These character sets can be stored in Big Endian or Little Endian style.

In a multilingual, heterogeneous environment, it it inevitable that different files will be written using different character sets. It is even possible that the same file might have different parts that use different character sets. How is it possible to tag these files internally so that Ω can apply the right translations?

Ω has two basic modes of input: the old TeX style, or the automatic Ω style. The old TeX style, is turned on when the `\noInputMode` command is read. The default mechanism is to use the automatic Ω style.

If the Ω style is being used, there are four modes, `ascii`, `ebcdic`, `unicode` (Big Endian) and `unicodeLE` (Little Endian), which correspond to the four situations above. Upon opening a file, Ω reads the first two characters. If the first character is hex `25` (`ascii %`), Ω assumes that the input character set is `ascii`. If the first character is hex `6c` (`ebcdic %`), Ω assumes that the input character set is `ebcdic`. If the first two characters form hex `0025` (Unicode `%`), Ω assumes that the input character set is Unicode. Finally, if the first two characters form hex `2500` (Unicode `%`), Ω assumes that the input character set is Unicode, Little Endian format. If none of these four situations occurs, then the default input mode is assumed.

Here are the instructions for specifying modes. All of these instructions apply only after the carriage return terminating the current input line. The `\inputMode` *mode* command, where *mode* is one of `ascii`, `ebcdic`, `unicode` or `unicodeLE`, states that after the carriage return, the input mode is *mode*. The `\noInputMode` command states that the old TeX style should be used. The `\defaultInputMode` *mode* instruction states that the default mode — when there is no comment character at the beginning of a file — should be *mode*. As for `\noDefaultInputMode`, it states that there is no default mode, and that whatever settings existed when opening the file should remain.

The default mode when the system begins is Ω style, assuming `ascii`. This is sufficient for all the `iso-8859` character sets, many national character sets, and most mixed-length character sets used in East Asia.

Once the basic family of character sets has been determined, Ω can read the files, and actually interpret control sequences. It is then possible to be more specific and to specify exactly what translation process must be applied to the entire file to convert the input to Unicode.

For the moment, input translations are simply single ΩCPs, which differ from input filters in that they apply to *all* characters in a file, not simply the letters and other characters in horizontal mode. For each kind of mode, there can be a default input translation.

As for the mode instructions, each instruction only applies after the carriage return terminating the current line. The `\inputTranslation` *ocp* command states that after this line, all input will be passed through translation process *ocp*. The `\noInputTranslation` states that no input will be translated.

The `\defaultAsciiInputTranslation` *ocp*, `\defaultEbcdicInputTranslation` *ocp* and `\defaultUnicodeInputTranslation` *ocp* commands state what the default translations will be for each of the modes. Finally, the `\noDefaultAsciiInputTranslation`, `\noDefaultEbcdicInputTranslation`, `\noDefaultUnicodeInputTranslation`, and `\noDefaultUnicodeLEInputTranslation` commands remove default translations.

Upon startup, there is no default translation for `ascii` or `unicode` modes, but there is one for `ebcdic`, namely

```
\ocp\OCPebcdic=ebcdic
\defaultEbcdicInputTranslation\OCPebcdic
```

13 Further work

The Ω project is far from finished. Currently much of the current work is geared towards font development. Nevertheless, new functionality is to be added in the future. In particular, more general methods for hyphenation, as well as for text output, using ΩTPs, are envisaged.