HeVeA User Documentation

Version 2.34

Luc Maranget*

March 26, 2020

Abstract

HeVeA is a LaTeX to HTML translator. The input language is a fairly complete subset of LaTeX 2ε (old LaTeX style is also accepted) and the output language is HTML that is (hopefully) correct with respect to version 5.

HeVeA understands LaTeX macro definitions. Simple user style files are understood with little or no modifications. Furthermore, HeVeA customisation is done by writing LaTeX code.

HeVeA is written in Objective Caml, as many lexers. It is quite fast and flexible. Using HeVeA it is possible to translate large documents such as manuals, books, etc. very quickly. All documents are translated as one single HTML file. Then, the output file can be cut into smaller files, using the companion program HaCha.

HeVeA can also be instructed to output plain text or info files.

Information on HeVeA is available at `http://hevea.inria.fr/`.

*Inria Paris – CS 42112, 75589 Paris Cedex 12. `Luc.Maranget@inria.fr`

Contents

Part A
Tutorial

1 How to get started

Assume that you have a file, `a.tex`, written in LaTeX, using the `article`, `book` or `report` style. Then, translation is achieved by issuing the command:

```
# hevea a.tex
```

Probably, you will get some warnings. If HeVeA does not crash, just ignore them for the moment (Section 4 explains how to correct errors).

If everything goes fine, this will produce a new file, `a.html`, which you can visualise through a HTML browser.

If you wish to experiment HeVeA on small LaTeX source fragments, then launch HeVeA without arguments. HeVeA will read its standard input and print the translation on its standard output. For instance:

```
# hevea
$x \in \mathcal{E}$
^D
<span style="font-style:italic">x</span> &#X2208; <span style="color:red"><span style="font-style:itali
```

Incidentally, notice that the symbol "$\in$" translates to the appropriate numerical character reference and that the calligraphic letter "$\mathcal{E}$" renders as a red "E". You can find some more elaborate examples[1] in the on-line documentation.

2 Style files

LaTeX style files are files that are not intended to produce output, but define document layout parameters, commands, environments, etc.

2.1 Standard base styles

The base style of a LaTeX document is the argument to the `\documentclass` command (`\documentstyle` in old style). Normally, the base style of a document defines the structure and appearance of the whole document.
HeVeA really knows about two LaTeX base styles, `article` and `book`. Additionally, the `report` base style is recognized and considered equivalent to `book` and the `seminar` base style for making slides is recognized and implemented by small additions on the `article` style.

Base style *style* is implemented by an HeVeA specific style file *style*`.hva`. More precisely, HeVeA interprets `\documentclass{`*style*`}` by attempting to load the file *style*`.hva` (see section C.1.1.1 on where HeVeA searches for files). Thus, at the moment, HeVeA distribution includes the files, `article.hva`, `book.hva`, etc.

2.2 Other base styles

Documents whose base style is not recognized by HeVeA can be processed when the unknown base style is a derivation of a recognized base style.

Let us assume that `doc.tex` uses an exotic base style such as `acmconf`. Then, typing `hevea doc.tex` will yield an error, since HeVeA cannot find the `acmconf.hva` file:

[1] `http://hevea.inria.fr/examples/index.html`

```
# hevea.opt doc.tex
doc.tex:1: Warning: Cannot find file: acmconf.hva
doc.tex:1: Error while reading LaTeX: No base style
Adios
```

This situation is avoided by invoking HeVeA with the known base style file `article.hva` as an extra argument:

```
# hevea article.hva doc.tex
```

The extra argument instructs HeVeA to load its `article.hva` style file before processing `doc.tex`. It will then ignore the document base style specified by `\documentclass` (or `\documentstyle`).

Observe that the fix above works because the `acmconf` and `article` base styles look the same to the document (*i.e.* they define the same macros). More generally, most base styles that are neither *article* nor *book* are in fact variations on either two of them. However, such styles usually provides extra macros. If users documents use these macros, then users should also instruct HeVeA about them (see section 4.1).

Finally, it is important to notice that renaming a base style file *style*`.cls` into *style*`.hva` will not work in general. As a matter of fact, base style files are TeX and not LaTeX source and HeVeA will almost surely fail on TeX-ish input.

2.3 Other style files

A LaTeX document usually loads additional style files, by using the commands `\input` or `\usepackage` or `\input`.

2.3.1 Files loaded with `\input`

Just like LaTeX, HeVeA reacts to the construct `\input{`*file*`}` by loading the file *file*. (if I got it right, HeVeA even follows TeX's crazy conventions on `.tex` extensions).

As it is often the case, assume that the document `doc.tex` has a `\input{mymacros.tex}` instruction in its preamble, where `mymacros.tex` gathers custom definitions. Hopefully, only a few macros give rise to trouble: macros that performs fine typesetting or TeXish macros. Such macros need to be rewritten, using basic LaTeX constructs (section 4 gives examples of macro-rewriting). The new definitions are best collected in a style file, `mymacros.hva` for instance. Then, `doc.tex` is to be translated by issuing the command:

```
# hevea mymacros.hva doc.tex
```

The file `mymacros.hva` is processed before `doc.tex` (and thus before `mymacros.tex`). As a consequence of HeVeA behaviour with respect to definition and redefinition (see section B.8.1), the macro definitions in `mymacros.hva` take precedence over the ones in `mymacros.tex`, provided the document original definitions (the ones in `mymacros.tex`) are performed by `\newcommand` (or `\newenvironment`).

Another situation is when HeVeA fails to process a whole style file. Usually, this means that HeVeA crashes on that style file. The basic idea is then to write a `mymacros.hva` style file that contains alternative definitions for all the commands defined in `mymacros.sty`. Then, HeVeA should be instructed to load `mymacros.hva` and not to load `mymacros.tex`. This is done by invoking `hevea` as follows:

```
# hevea mymacros.hva -e mymacros.tex doc.tex
```

Of course, `mymacros.hva` must now contain replacements for all the useful macros of `mymacro.tex`.

2.3.2 Files loaded with `\usepackage`

As far as I know, LaTeX reacts to the construct `\usepackage{`*name*`}` by loading the file *name*`.sty`. HeVeA reacts in a similar, but different, manner, by loading the file *name*`.hva`.

HeVeA distributions already includes quite a few `.hva` implementations of famous packages (see section B.17). When a given package (say `zorglub`) is not implemented, the situation may not be as bad as it may seem first. Hopefully, you are only using a few commands from package `zorglub`, and you feel confident enough to implement them yourself. Then, it suffices to put your definitions in file `zorglub.hva` and HeVeA will react to `\usepackage{zorglub}` by loading `zorglub.hva`.

See section B.5.2 for the full story on `\usepackage`.

3 A note on style

3.1 Spacing, Paragraphs

Sequence of spaces normally are translated into one single space. Newlines in the input document undergo a special treatement. A newline triggers a special scanning mode that reads all following spaces and newlines. In case at least one additional newline character is read, then HeVeA executes the `\par` command. Otherwise, HeVeA outputs a single newline character. This process approximates TeX process for introducting paragraph breaks and, as a result, empty lines produce paragraph breaks.

Space after commands with no argument is skipped (as in LaTeX) — however this is not true in math mode, as explained in section 3.2.1.

The following two subsections describe management of paragraphs and spaces after command sequences in greater detail. They can be skipped in first reading.

3.1.1 Spurious Paragraphs

Paragraphs are rendered by the means of `p` elements. HeVeA is a bit simplistic in breaking paragraphs and spurious paragraphs may be present in the final HTML document. Normally, as HeVeA never outputs `p` elements whose contents is made of spaces only, this should not happen very often. Unfortunately, some commands do not produce any output in LaTeX, while they do produce output in HeVeA: those commands are `\label`, `\index` etc. HeVeA translates `\label{`*name*`}` into the anchor `<a id="`*name*`"></a>`. As a result, the following source fragment will introduce a spurious paragraph.

```
This a first paragraph.

\label{label}

This is another paragraph.
```

Indeed, whe have the following translation:

```
<p>This a first paragraph.</p>
<p><a id="label"></a></p>
<p>This is another paragraph.</p>
```

Most of the time, such extra paragraphs remain unnoticed. Of course, they can be supressed by erasing one of the empty lines. For instance:

```
This a first paragraph.

\label{label}
This is another paragraph.
```

A similar situation occurs when a sectioning command is followed by `\label` and a paragraph break:

```
\section*{A section}\label{section:label}

First paragraph.
```

Produced HTML is, after a few cosmetic simplifications:

```
<h2 class="section">A section</h2>
<p><a id="section:label"></a></p>
<p>First paragraph.</p>
```

Output is so, because closing the element `h2` implies re-opening a new paragraph. Here, two possible re-writing of source are:

```
\section*{A\label{section:label} section}
First paragraph.
```

```
\section*{A section}
\label{section:label}First paragraph.
```

In all cases, this amounts to avoiding a paragraph whose contents consists in a sole `\label` command.

Spurious paragraphs are more easily seen by running `hevea` with the command-line option `-dv`, which instructs `hevea` to add border on some of the elements it produces, including `p` elements.

3.1.2 Spaces after Commands

Space after commands with no argument is skipped. Consider the following example:

```
\newcommand{\open}{(}
\newcommand{\close}{)}
\open text opened by ``\verb+\open+''
and closed by ``\verb+\close+''\close.
```

We get:

(text opened by "`\open`" and closed by "`\close`").

In the output above, the space after `\open` does not find its way to the output.

More generally, HEVEA tries to emulate LATEX behaviour in all situations, but discrepancies probably exist. Thus, users are invited to make explicit what they want. This is good practice anyway, because LATEX is mysterious here. Consider the following example, where the `\tryspace` macro is first applied and then expansed by hand:

```
\newcommand{\bfsymbol}{\textbf{symbol}}
\newcommand{\tryspace}[1]{#1 XXX}

Some space: \tryspace{\bfsymbol}\\
No space: \bfsymbol XXX
```

Spacing is a bit chaotic here, the space after **symbol** remains when `#1` is substituted for it by LATEX (or HEVEA).

Some space : **symbol** XXX
No space : **symbol**XXX

Note that, if a space before "XXX" is wanted, then one should probably write:

```
\newcommand{\tryspace}[1]{#1{} XXX}
```

Finally, whether the tabulation character is a space or not is random, so avoid tabs in your source document.

3.2 Math mode

HeVeA math mode is not very far from normal text mode, except that all letters are shown in italics and that space after macros is echoed.

However, typesetting math formulas in HTML rises two difficulties. First, formulas contain symbols, such as Greek letters; second, even simple formulas do not follow the simple basic typesetting model of HTML.

3.2.1 Spacing in math mode

By contrast with LaTeX, spaces from the input are significant in math mode, this feature allows users to instruct HeVeA on how to put space in their formulas. For instance, `\alpha\rightarrow\beta` is typeset without spaces between symbols, whereas `\alpha \rightarrow \beta` produces these spaces. Note that LaTeX ignores spaces in math mode, so that users can freely adjust HeVeA output without changing anything to LaTeX output.

3.2.2 Symbols

Figure 1: Some symbols

`\in`:	$\in$	`\notin`:	$\notin$
`\int`:	$\int$	`\prod`:	$\prod$
`\preceq`:	$\preceq$	`\prec`:	$\prec$
`\leq`:	$\leq$	`\geq`:	$\geq$
`\cup`:	$\cup$	`\cap`:	$\cap$
`\supset`:	$\supset$	`\subset`:	$\subset$
`\supseteq`:	$\supseteq$	`\subseteq`:	$\subseteq$

With respect to previous versions of HeVeA since the begining, the treatment of symbols has significantly evolved. Outputting symbols is now performed by using Unicode character references, an option that much more complies whith standards than the previous option of selecting a "symbol" font. Observe that this choice is now possible, because more and more browsers correctly display such references. See Figure 1 for a few such symbols.

However, this means that ancient or purposely limited browsers (such as text-oriented browsers) cannot display maths, as translated by HeVeA. For authors that insist on avoiding symbols that cannot be shown by any browser, HeVeA offers a degraded mode that outputs text in place of symbols. HeVeA operates in this mode when given the `-textsymbols` command-line option. Replacement text is in English. For instance. the "$\in$" symbol is replace by "in". This is far from being satisfactory, but degraded mode may be appropriate for documents than contain few symbols.

3.2.3 Displays

Apart from containing symbols, formulas specify strong typesetting constraints: sub-elements must be combined together following patterns that departs from normal text typesetting. For instance, fractions numerators and denominators must be placed one above the other. HeVeA handles such constraints in display mode only.

The main two operating modes of HeVeA are *text* mode and *display* mode. Text mode is the mode for typesetting normal text, when in this mode, text items are echoed one following the other and paragraph breaks are just blank lines, both in input and output. The so called *displayed-paragraph environments* of LaTeX (such as `center` or `quote`) are rendered by HTML block-level elements (such as `div` or `blockquote`). Rendering is correct becauses both LaTeX displayed environments and HTML block-level elements start a

new line. Conversly, since opening a HTML block-level elements means starting a new line, any text that sould appear inside a paragraph must be translated using only HTML text-level elements. HeVeA chooses to translate in-text formulas that way.

HeVeA display mode allows more control on text placement, since entering display mode means opening a HTML `table` element and that tables allow to control the relative position of their sub-elements. Displays come in two flavor, horizontal displays and vertical displays. An horizontal display is a one-row table, while a vertical display is a one-column table. These tables holds display sub-elements, displays sub-elements being centered vertically in horizontal display mode and horizontally in vertical display mode.

Display mode is first opened by opening a `displaymath` environment (e.g. by `$$` or `\[`). Then, sub-displays are opened by LaTeX constructs which require them. For instance, a displayed fraction (`\frac`) opens a vertical display.

The distinction between text and display modes clearly appears while typesetting math formulas. An in-text formula such as `$\int_1^2 xdx = \frac{3}{2}$` appears as: $\int_1^2 xdx = 3/2$, while the same formula has a better aspect in display mode:

$$\int_1^2 xdx = \frac{3}{2}$$

As a consequence, HeVeA is more powerful in display mode and formulas should be displayed as soon as they get a bit complicated. This rule is also true in LaTeX but it is more strict in HeVeA, since HTML capabilities to typeset formulas inside text are quite poor. In particular, it is not possible to get in-text "real" fractions or in-text limit-like subscripts.

Users should remember that HeVeA is not TeX or LaTeX and that HeVeA author neither is D. E. Knuth nor L. Lamport. Thus, some formulas may be rendered poorly. For instance, two fractions with different denominator and numerator height look strange.

$$\frac{1}{\sum_{i=0}^{N} U_i} = \frac{\sum_{i=0}^{N} U_i}{1}$$

The reason is that vertical displays in an horizontal display are HTML tables that always get centered in the vertical direction. Such a crude model cannot faithfully emulate any TeX box placement.

Users can get an idea on how HeVeA combines elements in display mode by giving the `-dv` command-line option, which instructs HeVeA to add borders to the `table` elements introduced by displays.

3.2.4 Arrays and display mode

By contrast with formulas, which HeVeA attempts to render with text-level elements only when they appear inside paragraphs, LaTeX arrays always translate to the block-level element `table`, thereby introducing non-desired line breaks before and after in-text arrays. As a consequence, in-text arrays yield an acceptable output, only while alone in a paragraph.

However, since in some sense, all HTML tables are displayed, the `array` and `tabular` environments implicitly open display mode, thus allowing a satisfactory typesetting of formulas in arrays. More precisely, array elements whose column format specification is `l`, `c` or `r` are typeset in display mode (see section B.10.2).

3.3 Warnings

When HeVeA thinks it cannot translate a symbol or construct properly, it issues a warning. This draws user attention onto a potential problem. However, rendering may be correct.

Note that all warnings can be suppressed with the `-s` (silent) option. When a warning reveals a real problem, it can often be cured by writing a specific macro. The next two sections introduce HeVeA macros, then section 4 describes how to proceed with greater detail.

3.4 Commands

Just like LaTeX, HeVeA can be seen as a macro language, macros are rewritten until no more expansion is possible. Then, either some characters (such as letters, integers...) are outputed or some internal operation (such as changing font attributes, or arranging text items in a certain manner) are performed.

This scheme favors easy extension of program capabilities by users. However, predicting program behaviour and correcting errors may prove difficult, since final output or errors may occur after several levels of macro expansion. As a consequence, users can tailor HeVeA to their needs, but it remains a subtle task. Nevertheless, happy LaTeX users should enjoy customizing HeVeA, since this is done by writing LaTeX code.

3.5 Style choices

LaTeX and HTML differ in many aspects. For instance, LaTeX allows fine control over text placement, whereas HTML does not. More symbols and font attributes are available in LaTeX than in HTML. Conversely, HTML has font attributes, such as color, which standard LaTeX has not.

Therefore, there are many situations where HeVeA just cannot render the visual effect of LaTeX constructions. Here some choices have to be made. For instance, calligraphic letters (`\mathcal`) are rendered in red.

If you are not satisfied with HeVeA rendering of text style declarations, then you can choose your own, by redefining the `\cal` macros, using `\renewcommand`, the macro redefinition operator of LaTeX. The key point is that you need not worry about HeVeA internals: just redefine the old-LaTeX style text-style declarations (*i.e.* `\it`, `\sc`, etc.) and everything should get fine:

```
\renewcommand{\sc}{\Huge}
\renewcommand{\cal}{\em}
```

(See sections 4 and 5 on how to make such changes while leaving your file processable by LaTeX, and section 10.2 for a more thorough descripton of customizing type styles).

Note that many of LaTeX commands and environments are defined in the `hevea.hva` file that HeVeA loads before processing any input. These constructs are written using LaTeX source code, in the end they invoke HeVeA internal commands.

Other LaTeX constructs, such as LaTeX key constructs or HeVeA internal commands (see section 8.3), that require special processing are defined in HeVeA source code. However, the vast majority of these definitions can be overridden by a redefinition. This may prove useless, since there is little point in redefining core constructs such as `\newcommand` for instance.

4 How to detect and correct errors

Most of the problems that occur during the translation of a given LaTeX file (say `trouble.tex`) can be detected and solved at the macro-level. That is, most problems induce a macro-related warning and can be solved by writing a few macros. The best place for these macros is an user style file (say `trouble.hva`) given as argument to HeVeA.

```
# hevea trouble.hva trouble.tex
```

By doing so, the macros written specially for HeVeA are not seen by LaTeX. Even better, `trouble.tex` is not changed at all.

A worth-mentiong alternative is inserting `\usepackage{trouble}` in the document preamble. Then, given HeVeA semantics for `\usepackage` (see Section B.5.2), HeVeA-specific commands should be placed in the file "`trouble.hva`" file, while LaTeX-specific commands should be placed in teh file "`trouble.sty`".

Of course, adapting a document to HeVeA processing will be easier if the LaTeX source is written in a generic style, using macros. Note that this style is recommended anyway, since it facilitates document maintenance.

4.1 HeVeA does not know a macro

Consider the following LaTeX source excerpt:

```
You can \raisebox{.6ex}{\em raise} text.
```

LaTeX typesets this as follows:

You can *raise* text.

Since HeVeA does not know about `\raisebox`, it incorrectly processes this input. More precisely, it first prints a warning message:

```
trouble.tex:34: Unknown macro: \raisebox
```

Then, it goes on by translating the arguments of `\raisebox` as if they were normal text. As a consequence some `.6ex` is finally found in the HTML output:

You can .6ex*raise* text.

To correct this, you should provide a macro that has more or less the effect of `\raisebox`. It is impossible to write a generic `\raisebox` macro for HeVeA, because of HTML limitations. However, in this case, the effect of `\raisebox` is to raise the box *a little*. Thus, the first, numerical, argument to `\raisebox` can be ignored in a private `\raisebox` macro defined in `trouble.hva`:

```
\newcommand{\raisebox}[2]{$^{\mbox{#2}}$}
```

Now, translating the document yields:

You can ^{raise} text a little.

Of course, this will work only when all `\raisebox` commands in the document raise text a little. Consider, the following example, where text is both raised a lowered a little:

```
You can \raisebox{.6ex}{\em raise}
or \raisebox{-.6ex}{\em lower} text.
```

Which LaTeX renders as follows:

You can ^{raise} or $_{lower}$ text.

Whereas, with the above definition of `\raisebox`, HeVeA produces:

You can ^{raise} or ^{lower} text.

A solution is to add a new macro definition in the `trouble.hva` file:

```
\newcommand{\lowerbox}[2]{$_{\mbox{#2}}$}
```

Then, `trouble.tex` itself has to be modified a little.

```
You can \raisebox{.6ex}{\em raise}
or \lowerbox{-.6ex}{\em lower} text.
```

HeVeA now produces a satisfying output:

You can ^{raise} or $_{lower}$ text.

Note that, for the document to remain LaTeX-processable, it should also contain the following definition for `\lowerbox`:

```
\newcommand{\lowerbox}[2]{\raisebox{#1}{#2}}
```

This definition can safely be placed anywhere in `trouble.tex`, since by HeVeA semantics for `\newcommand` (see section B.8.1) the new definition will not overwrite the old one.

4.2 HeVeA incorrectly interprets a macro

Sometimes HeVeA knows about a macro, but the produced HTML does not look good when seen through a browser. This kind of errors is detected while visually checking the output. However, HeVeA does its best to issue warnings when such situations are likely to occur.

Consider, for instance, this definition of `\blob` as a small black square.

```
\newcommand{\blob}{\rule[.2ex]{1ex}{1ex}}
\blob\ Blob \blob
```

Which LaTeX typesets as follows:

■ Blob ■

HeVeA always translates `\rule` as `<hr>`, ignoring size arguments. Hence, it produces the following, wrong, output:

Blob

We may not be particularily commited to a square blob. In that case, other small symbols would perfectly do the job of `\blob`, such as a bullet (`\bullet`). Thus, you may choose to give `\blob` a definition in `trouble.hva`:

```
\newcommand{\blob}{\bullet}
```

This new definition yields the following, more satisfying output:

• Blob •

In case we do want a square blob, there are two alternatives. We can have LaTeX typeset some subparts of the document and then to include them as images, section 6 explains how to proceed. We can also find a square blob somewhere in the variety of Unicode (or do I mean ISO 10646?) characters, and define `\blob` as a numerical character reference. Here, the character `U+02588` seems ok.

```
\newcommand{\blob}{\@print@u{X2588}}
```

However, beware that not all browsers display all of Unicode...

4.3 HeVeA crashes

HeVeA failure may have many causes, including a bug. However, it may also stem from a wrong LaTeX input. Thus, this section is to be read before reporting a bug...

4.3.1 Simple cases: LaTeX also crashes

In the following source, environments are not properly balanced:

```
\begin{flushright}
\begin{quote}
This is right-flushed quoted text.
\end{flushright}
\end{quote}
```

Such a source will make both LaTeX and HeVeA choke. HeVeA issues the following error message that shows the LaTeX environment that is not closed properly:

```
./trouble.tex:6: Environment nesting error: html: 'DIV' closes 'BLOCKQUOTE'
./trouble.tex:4: Latex environment 'quote' is pending
Adios
```

Thus, when HeVeA crashes, it is a good idea to check that the input is correct by running LaTeX on it.

4.3.2 Complicated cases

Unfortunately, HeVeA may crash on input that does not affect LaTeX. Such errors usually relate to environment or group nesting.

Consider for instance the following "optimized" version of a `quoteright` environment:

```
\newenvironment{quoteright}{\quote\flushright}{\endquote}

\begin{quoteright}
This a right-flushed quotation
\end{quoteright}
```

The `\quote` and `\flushright` constructs are intended to replace `\begin{quote}` and `\begin{flushright}`, while `\endquote` stands for `\end{quote}`. Note that the closing `\endflushright` is omitted, since it does nothing. LaTeX accepts such an input and produces a right-flushed quotation.

However, HeVeA usually translates LaTeX environments to HTML block-level elements and it *requires* those elements to be nested properly. Here, `\quote` translates to `<blockquote>`, `\flushright` translates to `<div class="flushright">` and `\endquote` translates to `</blockquote>`. At that point, HeVeA refuses to generate obviously non-correct HTML and it crashes:

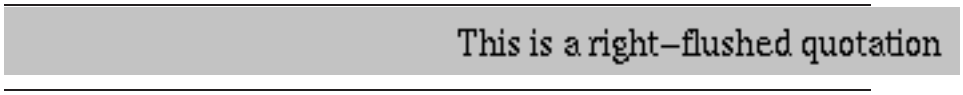
```
Giving up command: \@close
Giving up command: \endquote
Giving up command: \endquoteright
Giving up command: \end
./trouble.tex:7: Environment nesting error: html: 'BLOCKQUOTE' closes 'DIV'
./trouble.tex:5: Latex environment 'quoteright' is pending
Adios
```

Also notice that the error message above includes a backtrace showing the call-chain of commands.

In this case, the solution is easy: environments must be opened and closed consistently. LaTeX style being recommended, one should write:

```
\newenvironment{quoteright}
  {\begin{quote}\begin{flushright}}
  {\end{flushright}\end{quote}}
```

And we get:



Unclosed LaTeX groups ({...}) are another source of nuisance to HeVeA. Consider the following `horreur.tex` file:

```
\documentclass{article}

\begin{document}
In this sentence, a group is opened now {\em and never closed.
\end{document}
```

LaTeX accepts this file, although it produces a warning:

```
# latex horreur.tex
This is TeX, Version 3.14159 (Web2C 7.2)
  ...
(\end occurred inside a group at level 1)
Output written on horreur.dvi (1 page, 280 bytes).
```

By contrast, running HeVeA on `horreur.tex` yields a fatal error:

```
# hevea horreur.tex
Giving up command: \@raise@enddocument
Giving up command: \enddocument
Giving up command: \end
./horreur.tex:4: Environment nesting error: Latex env error: 'document' closes ''
./horreur.tex:3: Latex environment '' is pending
Adios
```

Thus, users should close opening braces where it belongs. Note that HeVeA error message "`Latex environment '`*env*`' is pending`" helps a lot in locating the brace that hurts.

4.3.3 Desperate cases

If HeVeA crashes on LaTeX source (not on TeX source), then you may have discovered a bug, or this manual is not as complete as it should. In any case, please report to `Luc.Maranget@inria.fr`.

To be useful, your bug report should include LaTeX code that triggers the bug (the shorter, the better) and mention HeVeA version number.

5 Making HeVeA and LaTeX both happy

A satisfactory translation from LaTeX to HTML often requires giving instructions to HeVeA. Typically, these instructions are macro definitions and these instructions should not be seen by LaTeX. Conversely, some source that LaTeX needs should not be processed by HeVeA. Basically, there are three ways to make input vary according to the processor, file loading, the `hevea` package and comments.

5.1 File loading

HeVeA and LaTeX treat files differently. Here is a summary of the main differences:

- LaTeX and HeVeA both load files given as arguments to `\input`, however when given the option `-e` *filename*, HeVeA does not load *filename*.
- HeVeA loads all files given as command-line arguments.
- Both LaTeX and HeVeA load style files given as optional arguments to `\documentstyle` and as arguments to `\usepackage`, but the files are searched by following different methods and considering different file extensions.

As a consequence, for having a file *latexonly* loaded by LaTeX only, it suffices to use `\input{`*latexonly*`}` in the source and to invoke HeVeA as follows:

`# hevea -e` *latexonly*...

Having *heveaonly* loaded by HeVeA only is more simple: it suffices to invoke HeVeA as follows:

`# hevea` *heveaonly*...

Finally, if one has an HeVeA equivalent *style*`.hva` for a LaTeX style file *style*`.sty`, then one should load the file as follows:

`\usepackage{`*style*`}`

This will result in, LaTeX loading *style*`.sty`, while HeVeA loads *style*`.hva`. As HeVeA will not fail in case *style*`.hva` does not exist, this is another method for having a style file loaded by LaTeX only.

Writing an HeVeA-specific file *file*`.hva` is the method of choice for supplying command definitions to HeVeA only. Users can then be sure that these definitions are not seen by LaTeX and will not get echoed to the *image* file (see section 6).

The file *file*`.hva` can be loaded by either supplying the command-line argument *file*`.hva`, or by `\usepackage{`*file*`}` from inside the document. Which method is better depends on whether you choose to override or to replace the document definition. In the command-line case, definitions from *file*`.hva` are processed before the ones from the document and will override them, provided the document definitions are made using `\newcommand` (or `\newenvironment`). In the `\usepackage` case, HeVeA loads *file*`.hva` at the place where LaTeX loads *file*`.sty`, hence the definitions from *file*`.hva` replace the definitions from *file*`.sty` in the strict sense.

5.2 The `hevea` package

The `hevea.sty` style file is intended to be loaded by LaTeX and not by HeVeA. It provides LaTeX with means to ignore or process some parts of the document. Note that HeVeA copes with the constructs defined in the `hevea.sty` file by default. It is important to notice that the `hevea.sty` style file from the distribution is a *package* in LaTeX 2_ε terms and that it is not compatible with old LaTeX. Moreover, the `hevea` package loads the `comment` package which must be present. Also notice that, for compatibility, HeVeA reacts to `\usepackage{hevea}` by loading its own version of the `comment` package (Section B.17.6).

5.2.1 Environments for selecting a translator

HeVeA and LaTeX perform the following actions on source inside the `latexonly`, `verblatex`, `htmlonly`, `rawhtml`, `toimage` and `verbimage` environments:

environment	HeVeA	LaTeX
`latexonly`	ignore, `\end{`*env*`}` constructs are processed (see section 5.2.2)	process
`verblatex`	ignore	process
`htmlonly`	process	ignore
`rawhtml`	echo verbatim (see section 8.4)	ignore
`toimage`	send to the *image* file, `\end{`*env*`}` constructs and macro characters are processed (see section 6)	process
`verbimage`	send to the *image* file (see section 6)	process

As an example, this is how some text can be typeset in purple by HeVeA and left alone by LaTeX:

```
We get:
\begin{htmlonly}%
\purple purple rain, purple rain%
\end{htmlonly}
\begin{latexonly}%
purple rain, purple rain%
\end{latexonly}%
\ldots
```

We get: purple rain, purple rain...

It is impossible to avoid the spurious space in HeVeA output for the source above. This extra spaces comes from the newline character that follows `\end{htmlonly}`. Namely this construct must appear in a line of its own for LaTeX to recognize it. Anyway, better control over spaces can be achieved by using the `hevea` boolean register or comments, see sections 5.2.3 and 5.3.

Also note that environments define a scope and that style changes (and non-global definitions) are local to them. For instance, in the example above, "..." appears in black in HTML output. However, as an exception, the environments `image` and `verbimage` do not create scope. It takes a little practice of HeVeA to understand why this is convenient.

5.2.2 Why are there two environments for ignoring input?

Some scanning and analysis of source is performed by HeVeA inside the `latexonly` environment, in order to allow `latexonly` to dynamically occur inside other environments.

More specifically, `\end{`*env*`}` macros are recognized and their *env* argument is tested against the name of the environment whose opening macro `\`*env* opened the `latexonly` environment. In that case, macro expansion of `\end`*env* is performed and any further occurrence of `\end{`*env'*`}` is tested and may get expanded if it matches a pending `\begin{`*env'*`}` construct.

This enables playing tricks such as:

```
\newenvironment{latexhuge}
  {\begin{latexonly}\huge}
  {\end{latexonly}}

\begin{latexhuge}
This will appear in huge font in \LaTeX{} output only.
\end{latexhuge}
```

LaTeX output will be:

This will appear in huge font in LaTeX output only.

While there is no HeVeA output.

Since HeVeA somehow analyses input that is enclosed in the `latexonly` environment, it may choke. However, this environment is intended to select processing by LaTeX only and might contain arbitrary source code. Fortunately, it remains possible to have input processed by LaTeX only, regardless of what it is, by enclosing it in the `verblatex` environment. Inside this environment, HeVeA performs no other action than looking for `\end{verblatex}`. As a consequence, the `\begin{verblatex}` and `\end{verblatex}` constructs may only appear in the main flow of text or inside the same macro body, a bit like LaTeX `verbatim` environment.

Relations between `toimage` and `verbimage` are similar. Additionally, formal parameters `#`i are replaced by actual arguments inside the `toimage` environment (see end of section 6.3 for an example of this feature).

5.2.3 The `hevea` boolean register

Boolean registers are provided by the `ifthen` package (see [LaTeX, Section C.8.5] and section B.8.5 in this document). Both the `hevea.sty` style file and HeVeA define the boolean register `hevea`. However, this register initial value is *false* for LaTeX and *true* for HeVeA.

Thus, provided, both the `hevea.sty` style file and the `ifthen` packages are loaded, the "purple rain" example can be rephrased as follows:

```
We get:
{\ifthenelse{\boolean{hevea}}{\purple}{}purple rain, purple rain}\ldots
```

We get: purple rain, purple rain...

Another choice is using the TeX-style conditional macro `\ifhevea` (see Section B.16.1.4):

```
We get:
{\ifhevea\purple\fi purple rain, purple rain}\ldots
```

We get: purple rain, purple rain...

5.3 Comments

HeVeA processes all lines that start with `%HEVEA`, while LaTeX treats these lines as comments. Thus, this is a last variation on the "purple rain" example:

```
We get
%HEVEA{\purple
purple rain, purple rain%
%HEVEA}%
\ldots
```

(Note how comments are placed at the end of some lines to avoid spurious spaces in the final output.)

We get: purple rain, purple rain...

Comments thus provide an alternative to loading the `hevea` package. For user convenience, comment equivalents to the `latexonly` and `toimage` environment are also provided:

environment	comment equivalent

`\begin{latexonly}... \end{latexonly}`	`%BEGIN LATEX`
	`...`
	`%END LATEX`

`\begin{toimage}... \end{toimage}`	`%BEGIN IMAGE`
	`...`
	`%END IMAGE`

Note that LaTeX, by ignoring comments, naturally performs the action of processing text between `%BEGIN...` and `%END...` comments. However, no environment is opened and closed and no scope is created while using comment equivalents.

6 With a little help from LaTeX

Sometimes, HeVeA just cannot process its input, but it remains acceptable to have LaTeX process it, to produce an image from LaTeX output and to include a link to this image into HeVeA output. HeVeA provides a limited support for doing this.

6.1 The *image* file

While outputting *doc*`.html`, HeVeA echoes some of its input to the *image* file, *doc*`.image.tex`. Part of this process is done at the user's request. More precisely, the following two constructs send *text* to the *image* file:

`\begin{toimage}`
text
`\end{toimage}`

`%BEGIN IMAGE`
text
`%END IMAGE`

Additionally, `\usepackage` commands, top-level and global definitions are automatically echoed to the image file. This enables using document-specific commands in *text* above.

Output to the image file builds up a current page, which is flushed by the `\imageflush` command. This command has the following effect: it outputs a strict page break in the *image* file, increments the image counter and output a `<img src="`*pagename*`.png">` tag in HeVeA output file, where *pagename* is build from the image counter and HeVeA output file name. Then the `imagen` script has to be run by:

`# imagen` *doc*

This will process the *doc*`.image.tex` file through LaTeX, `dvips`, `ghostscript` and a few others tools, which must all be present (see section C.4.1), finally producing one *pagename*`.png` file per page in the *image* file.

The usage of `imagen` is described at section C.1.5. Note that `imagen` is a simple shell script. Unix users can pass `hevea` the command-line option `-fix`. Then `hevea` will itself call `imagen`, when appropriate.

6.2 A toy example

Consider the "blob" example from section 4.2. Here is the active part of a `blob.tex` file:

```
\newcommand{\blob}{\rule[.2ex]{1ex}{1ex}}
\blob\ Blob \blob
```

This time, we would like `\blob` to produce a small black square, which `\rule[.2ex]{1ex}{1ex}` indeed does in LaTeX. Thus we can write:

```
\newcommand{\blob}{%
\begin{toimage}\rule[.2ex]{1ex}{1ex}%
\end{toimage}%
\imageflush}
\blob\ Blob \blob
```

Now we issue the following two commands:

```
# hevea blob.tex
# imagen blob
```

And we get:

Observe that the trick can be used to replace missing symbols by small `.png` images. However, the cost may be prohibitive, text rendering is generally bad, fine placement is ignored and font style changes are problematic. Cost can be lowered using `\savebox`, but the other problems remain.

6.3 Including Postscript images

In this section, a technique to transform included Postscript images into included bitmap images is described. Note that this technique is used by HEVEA implementation of the `graphics` package (see section B.14.1), which provides a more standard manner to include Postscript images in LATEX documents.

Included images are easy to manage: it suffices to let LATEX do the job. Let `round.ps` be a Postscript file, which is included as an image in the source file `round.tex` (which must load the *epsf* package):

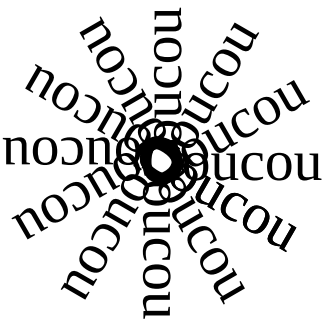
```
\begin{center}
\epsfbox{round.ps}
\end{center}
```

Then, HEVEA can have this image translated into a inlined (and centered) `.png` image by modifying source as follows:

```
\begin{center}
%BEGIN IMAGE
\epsfbox{round.ps}
%END IMAGE
%HEVEA\imageflush
\end{center}
```

(Note that the `round.tex` file still can be processed by LATEX, since comment equivalents of the `toimage` environment are used and that the `\imageflush` command is inside a `%HEVEA` comment — see section 5.3.)

Then, processing `round.tex` through HEVEA and `imagen` yields:



It is important to notice that things go smoothly because the `\usepackage{epsf}` command gets echoed to the *image* file. In more complicated cases, LaTeX may fail on the *image* file because it does not load the right packages or define the right macros.

However, the above solution implies modifying the original LaTeX source code. A better solution is to define the `\epsfbox` command, so that HeVeA echoes `\epsfbox` and its argument to the image file and performs `\imageflush`:

```
\newcommand{\epsfbox}[1]{%
\begin{toimage}
\epsfbox{#1}
\end{toimage}
\imageflush}
```

Such a definition must be seen by HeVeA only. So, it is best put in a separate file whose name is given as an extra argument on HeVeA command-line (see section 5.1). Putting it in the document source protected inside an `%HEVEA` comment is a bad idea, because it might then get echoed to the *image* file and generate trouble when LaTeX is later run by `imagen`.

Observe that the above definition of `\epsfbox` is a definition and not a redefinition (*i.e.* `\newcommand` is used and not `\renewcommand`), because HeVeA does not know about `\epsfbox` by default. Also observe that this not a recursive definition, since commands do not get expanded inside the `toimage` environment.

Finally, if the Postscript image is produced from a bitmap, it is a pity to translate it back into a bitmap. A better idea is first to generate a PNG file from the bitmap source independantly and then to include a link to that PNG file in HTML output, see section 8.2 for a description of this more adequate technique.

6.4 Using filters

Some programs extend LaTeX capabilities using a filter principle. In such a scheme, the document contains source fragments for the program. A first run of the program on LaTeX source changes these fragments into constructs that LaTeX (or a subsequent stage in the paper document production chain, such as `dvips`) can handle. Here again, the rule of the game is keeping HeVeA away from the normal process: first applying the filter, then making HeVeA send the filter output to the *image* file, and then having `imagen` do the job.

Consider the `gpic` filter, for making drawings. Source for `gpic` is enclosed in `.PS`...`.PE`, then the result is available to subsequent LaTeX source as a TeX box `\box\graph`. For instance the following source, from a `smile.tex` file, draws a "Smile!" logo as a centered paragraph:

```
.PS
ellipse "{\Large\bf Smile!}"
.PE
\begin{center}
~\box\graph~
\end{center}
```

Both the image description (`.PS... .PE`) and usage (`\box\graph`) are for the *image* file, and they should be enclosed by `%BEGIN IMAGE... %END IMAGE` comments. Additionally, the image link is put where it belongs by an `\imageflush` command:

```
%BEGIN IMAGE
.PS
ellipse "{\Large\bf Smile!}"
.PE
%END IMAGE
\begin{center}
%BEGIN IMAGE
~\box\graph~
%END IMAGE
%HEVEA\imageflush
\end{center}
```

The `gpic` filter is applied first, then come `hevea` and `imagen`:

```
# gpic -t < smile.tex > tmp.tex
# hevea tmp.tex -o smile.html
# imagen smile
```

And we get:

Observe how the `-o` argument to HeVeA is used and that `imagen` argument is HeVeA output basename (see section C.1.1.2 for the full definition of HeVeA output basename).

In the `gpic` example, modifying user source cannot be totally avoided. However, writing in a generic style saves typing. For instance, users may define the following environment for centered `gpic` pictures in LaTeX:

```
\newenvironment{centergpic}{}{\begin{center}~\box\graph~\end{center}}
```

Source code will now be as follows:

```
\begin{centergpic}
.PS
ellipse "{\Large\bf Smile!}"
.PE
\end{centergpic}
```

HeVeA will process this source correctly, provided it is given its own definition for the `centergpic` environment beforehand:

```
\newenvironment{centergpic}
  {\begin{toimage}}
  {\box\graph\end{toimage}\begin{center}\imageflush\end{center}}
```

Assuming that the definition above is in a smile.hva file, the command sequence for translating smile.tex now is:

```
# gpic -t < smile.tex > tmp.tex
# hevea smile.hva tmp.tex -o smile.html
tmp.tex:5: Warning: ignoring definition of \centergpic
tmp.tex:5: Warning: not defining environment centergpic
# imagen smile
```

The warnings above are normal: they are issued when HeVeA runs across the LaTeX-intended definition of the `centergpic` environment and refuses to override its own definition for that environment.

7 Cutting your document into pieces with HaCha

HeVeA outputs a single `.html` file. This file can be cut into pieces at various sectional units by HaCha

7.1 Simple usage

First generate your HTML document by applying HeVeA:

`# hevea` *doc*`.tex`

Then cut *doc*`.html` into pieces by the command:

`# hacha` *doc*`.html`

This will generate a simple root file `index.html`. This root file holds document title, abstract and a simple table of contents. Every item in the table of contents contains a link to or into a file that holds a "cutting" sectional unit. By default, the cutting sectional unit is *section* in the *article* style and *chapter* in the *book* style. The name of those files are *doc*001`.html`, *doc*002`.html`, etc.

Additionally, one level of sectioning below the cutting unit (*i.e.* subsections in the *article* style and sections in the *book* style) is shown as an entry in the table of contents. Sectional units above the cutting section (*i.e.* parts in both *article* and *book* styles) close the current table of contents and open a new one. Cross-references are properly handled, that is, the local links generated by HeVeA are changed into remote links.

The name of the root file can be changed using the `-o` option:

`# hacha -o root.html` *doc*`.html`

Some of HeVeA output get replicated in all the files generated by HaCha. Users can supply a header and a footer, which will appear at the begining and end of every page generated by HaCha. It suffices to include the following commands in the document preamble:

`\htmlhead{`*header*`}`
`\htmlfoot{`*footer*`}`

HaCha also makes every page it generates a clone of its input as regards attributes to the `<body ...>` opening tag and meta-information from the `<head>`... `<\head>` block. See section B.2 for examples of this replication feature.

By contrast, style information specified in the `style` elements from rom the `<head>`... `<\head>` block is not replicated. Instead, all style definitions are collected into an external style sheet file whose name is *doc*`.css`, and all generated HTML files adopt *doc*`.css` as an external style sheet. It is important to notice that, since version 1.08, HeVeA produces a `style` element by itself, even if users do not explicitely use styles. As a consequence, HaCha normally produces a file *doc*`.css`, which should not be forgotten while copying files to their final destination after a run of HaCha.

7.2 Advanced usage

HACHA behaviour can be altered from the document source, by using a counter and a few macros.

A document that explicitly includes cutting macros still can be typeset by LATEX, provided it loads the `hevea.sty` style file from the HEVEA distribution. (See section 5 for details on this style file). An alternative to loading the `hevea` package is to put all cutting instructions in comments starting with `%HEVEA`.

7.2.1 Principle

HACHA recognizes all sectional units, ordered as follows, from top to bottom: *part*, *chapter*, *section*, *subsection*, *subsubsection*, *paragraph* and *subparagraph*.

At any point between `\begin{document}` and `\end{document}`, there exist a current cutting sectional unit (cutting unit for short), a current cutting depth, a root file and an output file. Table of contents output goes to the root file, normal output goes to the output file. Cutting units start a new output file, whereas units comprised between the cutting unit and the cutting units plus the cutting depth add new entries in the table of contents.

At document start, the root file and the output file are HACHA output file (*i.e.* `index.html`). The cutting unit and the cutting depth are set to default values that depend on the document style.

7.2.2 Cutting macros

The following cutting instructions are for use in the document preamble. They command the cutting scheme of the whole document:

`\cuttingunit` This is a macro that holds the document cutting unit. You can change the default (which is *section* in the *article* style and *chapter* in the *book* style) by doing:

`\renewcommand{\cuttingunit}{`*secname*`}`.

`\tocnumber` Instruct HEVEA to put section numbers into table of content entries.

`\notocnumber` Instruct HEVEA *not* to put section numbers into table of content entries. This is the default.

`cuttingdepth` This is a counter that holds the document cutting depth. You can change the default value of 1 by doing `\setcounter{cuttingdepth}{`*numvalue*`}`. A cutting depth of zero means no other entries than the cutting units in the table of contents.

Other cutting instructions are to be used after `\begin{document}`. They all generate HTML comments in HEVEA output. These comments then act as instructions to HACHA.

`\cuthere{`*secname*`}{`*itemtitle*`}` Attempt a cut.

- If *secname* is the current cutting unit or the keyword `now`, then a new output file is started and an entry in the current table of contents is generated, with title *itemtitle*. This entry holds a link to the new output file.
- If *secname* is above the cutting unit, then the current table of contents is closed. The output file is set to the current root file.
- If *secname* is below the cutting unit and less than the cutting depth away from it, then an entry is added in the table of contents. This entry contains *itemtitle* and a link to the point where `\cuthere` appears.
- Otherwise, no action is performed.

`\cutdef[`*depth*`]{`*secname*`}` Open a new table of contents, with cutting depth *depth* and cutting unit *secname*. If the optional *depth* is absent, the cutting depth does not change. The output file becomes the root file. Result is unspecified if whatever *secname* expands to is a sectional unit name above the current cutting unit, is not a valid sectional unit name or if *depth* does not expand to a small positive number.

`\cutend` End the current table of contents. This closes the scope of the previous `\cutdef`. The cutting unit and cutting depth are restored. Note that `\cutdef` and `\cutend` must be properly balanced.

Commands `\cuthere` and `\cutend` have starred variants, which behave identically except for footnotes (see 7.3.7).

Default settings work as follows: `\begin{document}` performs

```
\cutdef*[\value{cuttingdepth}]{\cuttingunit}
```

and `\end{document}` performs `\cutend*`. All sectioning commands perform `\cuthere`, with the sectional unit name as first argument and the (optional, if present) sectioning command argument (*i.e.* the section title) as second argument. Note that starred versions of the sectioning commands also perform cutting instructions.

7.2.3 Table of links organisation

A table of links generated by HACHA is a list of links to generated files. Additionally, some sublists may be present, up to a certain depth. The items in those sublists are links inside generated files, they point to sectional unit titles below the cutting unit, up to a certain depth.

More precisely, let A be a certain sectional unit (*e.g.* "part"), let B be just below A (*e.g.* "section"), and let C be just below C (*e.g.* "subsection"). Further assume that cutting is performed at level B with a depth of more than one. Then, every unit A holds a one or several tables of links to generated files, and each generated file normally holds a B unit. Sublists with links to C units inside B units normally appear in the tables of links of level A. The command-line options `-tocbis` and `-tocter` instruct `hacha` to put sublists at other places. With `-tocbis` sublists are duplicated at the beginning of the B level files; while with `-tocter` sublist only appear at the beginning of the B level files.

In my opinion, default style is appropriate for documents with short B units; while `-tocbis` style is appropriate for documents with long B units with a few sub-units; and `-tocter` style is appropriate for documents with long B units with a lot of sub-units.

Whatever the style is, if a B unit is cut (*e.g.* because its text is enclosed in `\cutdef{`C`}`... `\cutend`), then every C unit goes into its own file and there is no sublist after the relevant B level entry in the A level table of links.

7.2.4 Examples

Consider, for instance, a *book* document with a long chapter that you want to cut at the section level, showing subsections:

```
\chapter{A long chapter}
.....

\chapter{The next chapter}
```

Then, you should insert a `\cutdef` at chapter start and a `\cutend` at chapter end:

```
\chapter{A long chapter}
%HEVEA\cutdef[1]{section}
.....
%HEVEA\cutend
\chapter{The next chapter}
```

Then, the file that would otherwise contain the long chapter now contains the chapter title and a table of sections. No other change is needed, since the command `\section` already performs the appropriate `\cuthere{section}{...}` commands, which were ignored by default. (Also note that cutting macros are placed inside `%HEVEA` comments, for LaTeX not to be disturbed).

The `\cuthere` macro can be used to put some document parts into their own file. This may prove appropriate for long cover pages or abstracts that would otherwise go into the root file. Consider the following document:

```
\documentclass{article}

\begin{document}

\begin{abstract} A big abstract \end{abstract}
...
```

Then, you make the abstract go to its own file as it was a cutting unit by typing:

```
\documentclass{article}
\usepackage{hevea}

\begin{document}
\cuthere{\cuttingunit}{Abstract}
\begin{abstract} A big abstract \end{abstract}
...
```

(Note that, this time, cutting macros appear unprotected in the source. However, LaTeX still can process the document, since the `hevea` package is loaded).

7.2.5 More and More Pages in Output

In some situations it may be appropriate to produce many pages from one source files. More specifically, loading the `deepcut` package will put all sectioning units of your document (from `\part` to `\subsection` in their own file.

Similarly, loading the `figcut` package will make all figures and tables go into their own file. The `figcut` package accepts two options, `show` and `noshow`. The former, which is the default, instructs HeVeA to repeat the caption into the main flow of text, with a link to the figure. The latter option disables the feature.

7.3 More Advanced Usage

In this section we show how to alter some details of HaCha behaviour. This includes controlling output file names and the title of generated web pages and introducing arbitrary cuts.

7.3.1 Controlling output file names

When invoked as `hacha` *`doc`*`.html`, HaCha produces a `index.html` table of links file that points into *doc*`001.html`, *doc*`002.html`, etc. content files. This is not very convenient when one wishes to point inside the document from outside. However, the `\cutname{`*name*`}` command sets the name of the current output file name as *name*.

Consider a document cut at the section level, which contains the following important section:

```
\section{Important\label{important} section}
...
```

To make the important section goes into file `important.html`, one writes:

```
\section{Important\label{important} section}\cutname{important.html}
...
```

Then, section "Important section" can be referenced from an HeVeA unaware HTML page by:

```
In this document, there is a very
<a href="important.html#important">important section</a>.
```

7.3.2 Controlling page titles

When HACHA creates a web page from a given sectional unit, the title of this page normally is the name of the sectional unit. For instance, the title of this very page should be "Cutting your document into pieces with HACHA". It is possible to insert some text at the beginning of all page titles, by using the `\htmlprefix` command. Hence, by writing `\htmlprefix{\hevea{} Manual: }` in the document, the title of this page would become: "HEVEA Manual: Cutting your document into pieces with HACHA" and the title of all other pages would show the same prefix.

7.3.3 Links for the root file

The command `\toplinks{`*prev*`}{`*up*`}{`*next*`}` instructs HACHA to put links to a "previous", "up" and "next" page in the root file. The following points are worth noticing:

- The `\toplink` command must appear in the document preamble (*i.e.* before `\begin{document}`).
- The arguments *prev*, *up* and *next* should expand to urls, notice that these argument are processed (see section 8.1.1).
- When one of the expected argument is left empty, the corresponding link is not generated.

This feature can prove useful to relate documents that are generated independently by HEVEA and HACHA.

7.3.4 Controlling link contents from the document

By default the links to the previous, up and next pages show a small icon (an appropriate arrow). This can be changed with the command `\setlinkstext{`*prev*`}{`*up*`}{`*next*`}`, where *prev*, *up* and *next* are some LaTeX source. For instance the default behaviour is equivalent to:

```
\setlinkstext
  {\imgsrc[alt="Previous"]{previous_motif.svg}}
  {\imgsrc[alt="Up"]{contents_motif.svg}}
  {\imgsrc[alt="Next"]{next_motif.svg}}
```

Command `\setlinkstext` behaves as `\toplinks` does. That is, it must occur in document preamble, arguments are processed and empty arguments yield no effect (*i.e.* defaults apply).

7.3.5 Complete control over navigation links

The previous commands only impact the contents of the navigation links. It is possible, although reserved to avanced users, to achieve greater control by using the `\formatlinks` command. The `\formatlinks` command takes four arguments which are command themselves. The last three command format the "previous", "up" and "next" links respectively, while the first argument formats the resulting group of links. For instance, one can avoid images and for arrows and typeset the full set of navivation links in a purple border (see Section 9 for styling techniques) as follows:

```
\newstyle{a.navarrow}{font-family:monospace;font-size:x-large;color:purple}
\newstyle{div.navarrows}{border:solid purple;display:inline-block;padding:1ex;}
\newcommand{\myprev}[1]{\ahref[class="navarrow" title="Previous" ]{#1}{$\rightarrow$}\quad}
\newcommand{\myup}[1]{\quad\ahref[class="navarrow" title="Up" ]{#1}{$\uparrow$}\quad}
\newcommand{\mynext}[1]{\quad\ahref[class="navarrow" title="Next" ]{#1}{$\writearrow$}}

\newcommand{\mylinks}[1]{\@open{div}{class="navarrows"}#1\@close{div}\end{center}}
\formatlinks{\mylinks}{\myprev}{\myup}{\mynext}
```

7.3.6 Cutting a document anywhere

Part of a document goes to a separate file when enclosed in a `cutflow` environment:

`\begin{cutflow}{`*title*`}`... `\end{cutflow}`

The content "..." will go into a file of its own, while the argument *title* is used as the title of the introduced HTML page.

The HTML page introduced here does not belong to the normal flow of text. Consequently, one needs an explicit reference from the normal flow of text into the content of the `cutflow` environment. This will occur naturally when the content of the `cutflow` environment. contains a `\label` construct. This look natural in the following quiz example:

```
\paragraph{A small quiz}
\begin{enumerate}
\item What is black?
\item What is white?
\item What is Dylan?
\end{enumerate}
Answers in section~\ref{answers}.
\begin{cutflow}{Answers}
\paragraph{Quiz answers}\label{answers}
\begin{enumerate}
\item Black is black.
\item White is white.
\item Dylan is Dylan.
\end{enumerate}
\end{cutflow}
```

However,introducing HTML hyperlink targets and references with the `\aname` and `\ahrefloc` commands (see section 8.1.1) will be more practical most of the time.

The starred variant environment `cutflow*` is the same as `cutflow`, save for the HTML header and footer (see Section 7.1) which are not replicated in the introduced page.

7.3.7 Footnotes

Footnote texts (given as arguments either to `\footnote` or `\footnotetext`) do not go directly to output. Instead, footnote texts accumulate internally in a *buffer*, awaiting to be flushed. The flushing of notes is controlled by the means of a current *flushing unit*, which is a sectional unit name or *document* — a fictional unit above all units. At any point, the current flushing unit is the value of the command `\@footnotelevel`. In practice, the flushing of footnote texts is performed by two commands:

- `\flushdef{`*secname*`}` simply sets the flushing unit to *secname*.
- `\footnoteflush{`*secname*`}` acts as follows:
  - If argument *secname* is equal to or above the current flushing unit, then footnote texts are flushed (if any). In the output, the texts themselves are surrounded by special comments that tag them as footnote texts and record *secname*.
  - Otherwise, no action is performed.

The *article* style file performs `\flushdef{document}`, while the *book* style file performs `\flushdef{chapter}`. At the end of processing, `\end{document}` performs `\footnoteflush{\@footnotelevel}`, so as to flush any pending notes.

Cutting commands interact with footnote flushing as follows:

- `\cuthere{`*secname*`}` executes `\footnoteflush{`*secname*`}`. Remember that all sectioning commands perform `\cuthere` with their sectional unit name as argument.
- `\cutdef{`*secname*`}` saves the current flushing unit and buffer on some internal stack, starts a new buffer for footnote texts, and sets the current flushing unit to *secname* (by performing `\flushdef{`*secname*`}`).
- `\cutend` first flushes any pending texts (by performing `\footnoteflush` with the current flushing unit as argument), and restores the flushing unit and footnote text buffer saved by the matching `\cutdef`.
- The starred variants `\cutdef*` and `\cutend*` perform no operation that is related to footnotes.

Later, when running across footnote texts in its input file, HACHA sometimes put notes in a separate file. More precisely, HACHA has knowledge of the current *cutting level*, the current sectional unit where cuts occur — as given by the relevant `\cutdef`. Moreover, HACHA knows the current *section level* — that is, the last sectional command processed. Besides, HACHA extracts the *note level* from the comments that surround the notes (as given by the command `\footnoteflush` that produced the notes). Then, HACHA creates a separate file for notes when the cutting level and the note level differ, or when the current level is above the cutting level (*e.g.* the current level is `document` while the cutting level is `chapter`). As a result, notes should stay where they are when they occur at the end of HACHA output file and otherwise go to a separate file.

To make a complicated story even more complicated, footnotes in `minipage` environments or in the arguments to `\title` or `\author` have a different, I guess satisfactory, behaviour.

Given the above description, footnotes are managed by default as follows.

- In style *article*, `hevea` puts all footnotes go at the end of the HTML file. A later run of `hacha` creates a separate footnote file.
- In style *book*, footnotes are collected at the end of chapters. A later run of `hacha` leaves them where they are. Footnotes in the title or author names are managed specially, they will normally appear at the end of the root file.

In case you wish to adopt a *book*-like behaviour for an *article* (footnotes at the end of sections), it suffices to insert `\flushdef{section}` in the document preamble.

8 Generating HTML constructs

HEVEA output language being HTML, it is normal for users to insert hypertext constructs their documents, or to control colours.

8.1 High-Level Commands

HEVEA provides high-level commands for generating hypertext constructs. Users are advised to use these commands in the first place, because it is easy to write incorrect HTML and that writing HTML directly may interfere in nasty ways with HEVEA internals.

8.1.1 Commands for Hyperlinks

A few commands for hyperlink management and included images are provided, all these commands have appropriate equivalents defined by the `hevea` package (see section 5.2). Hence, a document that relies on these high-level commands still can be typeset by LATEX, provided it loads the `hevea` package.

Macro	HeVeA	LaTeX
\ahref{*url*}{*text*}	make *text* an hyperlink to *url*	echo *text*
\footahref{*url*}{*text*}	make *text* an hyperlink to *url*	make *url* a footnote to *text*, *url* is shown in typewriter font
\ahrefurl{*url*}	make *url* an hyperlink to *url*.	typeset *url* in typewriter font
\ahrefloc{*label*}{*text*}	make *text* an hyperlink to *label* inside the document	echo *text*
\aname{*label*}{*text*}	make *text* an hyperlink target with label *label*	echo *text*
\mailto{*address*}	make *address* a "mailto" link to *address*	typeset *address* in typewriter font
\imgsrc[*attr*]{*url*}	insert *url* as an image, *attr* are attributes in the HTML sense	do nothing
\home{*text*}	produce a home-dir url both for output and links, output aspect is: "~*text*"	

It is important to notice that all arguments are processed. For instance, to insert a link to my home page, (`http://pauillac.inria.fr/~maranget/index.html`), you should do something like this:

```
\ahref{http://pauillac.inria.fr/\home{maranget}/index.html}{his home page}
```

Given the frequency of `~`, `#` etc. in urls, this is annoying. Moreover, the immediate solution, using `\verb`, `\ahref{\verb" ... /~maranget/..."}{his home page}` does not work, since LaTeX forbids verbatim formatting inside command arguments.

Fortunately, the `url` package provides a very convenient `\url` command that acts like `\verb` and can appear in other command arguments (unfortunately, this is not the full story, see section B.17.11). Hence, provided the `url` package is loaded, a more convenient reformulation of the example above is:

```
\ahref{\url{http://pauillac.inria.fr/~maranget/index.html}}{his home page}
```

Or even better:

```
\urldef{\lucpage}{\url}{http://pauillac.inria.fr/~maranget/index.html}
\ahref{\lucpage}{his home page}
```

It may seem complicated, but this is a safe way to have a document processed both by LaTeX and HeVeA. Drawing a line between url typesetting and hyperlinks is correct, because users may sometime want urls to be processed and some other times not. Moreover, HeVeA (optionally) depends on only one third party package: `url`, which is as correct as it can be and well-written.

In case the `\url` command is undefined at the time `\begin{document}` is processed, the commands `\url`, `\oneurl` and `\footurl` are defined as synonymous for `\ahref`, `\ahrefurl` and `\footahref`, thereby ensuring some compatibility with older versions of HeVeA. Note that this usage of `\url` is deprecated.

8.1.2 HTML style colours

Specifying colours both for LaTeX and HeVeA should be done using the `color` package (see section B.14.2). However,one can also specify text color using special type style declarations. The `hevea.sty` style file define no equivalent for these declarations, which therefore are for HeVeA consumption only.

Those declarations follow HTML conventions for colours. There are sixteen predefined colours:

```
\black, \silver, \gray, \white, \maroon, \red, \fuchsia, \purple,
\green, \lime, \olive, \yellow, \navy, \blue, \teal, \aqua
```

Additionally, the current text color can be changed by the declaration `\htmlcolor{`*number*`}`, where *number* is a six digit hexadecimal number specifying a color in the RGB space. For instance, the declaration `\htmlcolor{404040}` changes font color to dark gray,

8.2 More on included images

The `\imgsrc` command becomes handy when one has images both in Postscript and GIF (or PNG or JPG) format. As explained in section 6.3, Postscript images can be included in LaTeX documents by using the `\epsfbox` command from the `epsf` package. For instance, if `screenshot.ps` is an encapsulated Postscript file, then a `doc.tex` document can include it by:

```
\epsfbox{screenshot.ps}
```

We may very well also have a GIF version of the screenshot image (or be able to produce one easily using image converting tools), let us store it in a `screenshot.ps.gif` file. Then, for HeVeA to include a link to the GIF image in its output, it suffices to define the `\epsfbox` command in the `macro.hva` file as follows:

```
\newcommand{\epsfbox}[1]{\imgsrc{#1.gif}}
```

Then HeVeA has to be run as:

```
# hevea macros.hva doc.tex
```

Since it has its own definition of `\epsfbox`, HeVeA will silently include a link the GIF image and not to the Postscript image.

If another naming scheme for image files is preferred, there are alternatives. For instance, assume that Postscript files are of the kind *name*`.ps`, while GIF files are of the kind *name*`.gif`. Then, images can be included using `\includeimage{`*name*`}`, where `\includeimage` is a specific user-defined command:

```
\newcommand{\includeimage}[1]{\ifhevea\imgsrc{#1.gif}\else\epsfbox{#1.ps}\fi}
```

Note that this method uses the `hevea` boolean register (see section 5.2.3). If one does not wish to load the `hevea.sty` file, one can adopt the slightly more verbose definition:

```
\newcommand{\includeimage}[1]{%
%HEVEA\imgsrc{#1.gif}%
%BEGIN LATEX
\epsfbox{#1.ps}
%END LATEX
}
```

When the Postscript file has been produced by translating a bitmap file, this simple method of making a bitmap image and using the `\imgsrc` command is the most adequate. It should be preferred over using the more automated *image* file mechanism (see section 6), which will translate the image back from Postscript to bitmap format and will thus degrade it.

8.3 Internal macros

In this section a few of HeVeA internal macros are described. Internal macros occur at the final expansion stage of HeVeA and invoke Objective Caml code.

Normally, user source code should not use them, since their behaviour may change from one version of HeVeA to another and because using them incorrectly easily crashes HeVeA. However:

- Internal macros are almost mandatory for writing supplementary base style files.
- Casual usage is a convenient (but dangerous) way to finely control output (cf. the examples in the next section).
- Knowing a little about internal macros helps in understanding how HeVeA works.

The general principle of HeVeA is that LaTeX environments `\begin{`*env*`}`... `\end{`*env*`}` get translated into HTML block-level elements <*block attributes*>... </*block*>. More specifically, such block level elements are opened by the internal macro `\@open` and closed by the internal macro `\@close`. As a special case, LaTeX groups `{`... `}` get translated into HTML *groups*, which are shadow block-level elements with neither opening nor closing tag.

In the following few paragraphs, we sketch the interaction of `\@open`...`\@close` with paragraphs. Doing so, we intend to warn users about the complexity of the task of producing correct HTML, and to encourage them to use internal macros, which, most of the time, take nasty details into account.

Paragraphs are rendered by `p` elements, which are opened and closed automatically. More specifically, a first `p` is opened after `\begin{document}`, then paragraph breaks close the active `p` and open a new one. The final `\end{document}` closes the last `p`. In any occasion, paragraphs consisting only of space characters are discarded silently.

Following HTML "normative reference [HTML-5a]", block-level elements cannot occur inside `p`; more precisely, block-level opening tags implicitly close any active `p`. As a consequence, HeVeA closes the active `p` element when it processes `\@open` and opens a new `p` when it processes the matching `\@close`. Generally, no `p` element is opened by default inside block-level elements, that is, HeVeA does not immediately open `p` after having processed `\@open`. However, if a paragraph break occurs later, then a new `p` element is opened, and will be closed automatically when the current block is closed. Thus, the first "paragraph" inside block-level elements that include several paragraphs is not a `p` element. That alone probably prevents the consistent styling of paragraphs with style sheets.

Groups behave differently, opening or closing them does not close nor open `p` elements. However, processing paragraph breaks inside groups involves temporarily closing all groups up to the nearest enclosing `p`, closing it, opening a new `p` and finally re-opening all groups. Opening a block-level element inside a group, similarly involves closing the active `p` and opening a new `p` when the matching `\@close` is processed.

Finally, display mode (as introduced by `$$`) is also complicated. Displays basically are `table` elements with one row (`tr`), and HeVeA manages to introduce table cells (`td`) where appropriate. Processing `\@open` inside a display means closing the current cell, starting a new cell, opening the specified block, and then immediately opening a new display. Processing the matching `\@close` closes the internal display, then the specified block, then the cell and finally opens a new cell. In many occasions (in particular for groups), either cell break or the internal display may get cancelled.

It is important to notice that primitive arguments *are* processed (except for the `\@print` primitive, and for some of the basic style primitives). Thus, some characters cannot be given directly (e.g. `#` and `%` must be given as `\#` and `\%`).

`\@print{`*text*`}` Echo *text* verbatim. As a consequence use only ascii in *text*.

`\@getprint{`*text*`}` Process *text* using a special output mode that strips off HTML tags. This macro is the one to use for processed attributes of HTML tags.

`\@hr[`*attr*`]{`*width*`}{`*height*`}` Output an HTML horizontal rule, *attr* is attributes given directly (e.g. `SIZE=3 HOSHADE`), while *width* and *height* are length arguments given in the LaTeX style (e.g. `2pt` or `.5\linewidth`).

`\@print@u{`*n*`}` Output the (Unicode) character "*n*", which can be given either as a decimal number or an hexadecimal number prefixed by "`X`".

`\@open{`*block*`}{`*attributes*`}` Open HTML block-level element *block* with attributes *attributes*. The block name *block* **must** be lowercase. As a special case *block* may be the empty string, then a HTML *group* is opened.

`\@close{`*block*`}` Close HTML block-level element *block*. Note that `\@open` and `\@close` must be properly balanced.

`\@out@par{`*arg*`}` If occurring inside a `p` element, that is if a `<p>` opening tag is active, `\@out@par` first closes it (by emitting `</p>`), then formats *arg*, and then re-open a `p` element. Otherwise `\@out@par` simply formats *arg*. This command is adequate when formatting *arg* produces block-level elements.

Text-level elements are managed differently. They are not seen as blocks that must be closed explicitly. Instead they follow a "declaration" style, similar to the one of LaTeX "text-style declarations" — namely, `\itshape`, `\em` etc. Block-level elements (and HTML groups) delimit the effect of such declarations.

`\@span{`*attr*`}` Declare the text-level element `span` (with given attributes) as active. The text-level element `span` will get opened as soon as necessary and closed automatically, when the enclosing block-level elements get closed. Enclosed block-level elements are treated properly by closing `span` before them, and re-opening `span` (with given attributes) inside them. The following text-level constructs exhibit similar behaviour with respect to block-level elements.

`\@style{`*shape*`}` Declare the text shape *shape* (which must be lowercase) as active. Text shapes are known as font style elements (`i`, `tt`, etc.; **warning:** most of font style elements are depreciated in HTML5, and some of them are no longer valid, prefer CSS in `span` tags) or phrase elements (`em`, etc.) in the HTML terminology.

`\@styleattr{`*name*`}{`*attr*`}` This command generalises both `\@span` and `\@style`, as both a text-level element name *name* and attributes are specified. More specifically, `\@span{`*attr*`}` can be seen as a shorthand for `\@styleattr{span}{`*attr*`}`; while `\@style{`*name*`}` can be seen as a shorthand for `\@styleattr{`*name*`}{}`.

`\@fontsize{`*int*`}` Declare the text-level element `span` with attribute `style="font-size:`*font-size*`"` as active. The argument *int* must be a small integer in the range `1`,`2`, ... , `7`. `hevea` computes *font-size*, a CSS fontsize value, from *int*. More specifically, *font-size* will range from `x-small` to `120%` included in a `xx-large`, 3 being the default size `medium`. Notice that `\@fontsize` is deprecated in favour of `\@span` with proper fontsize declarations: `\@span{style="font-size=xx-small"}`, `\@span{style="font-size=x-small"}`, `\@span{style="font-size=small"}`, etc.

`\@fontcolor{`*color*`}` Declare the text-level element `span` with attribute `"style=`*color*`"` as active. The argument *color* must be a color attribute value in the HTML style. That is either one of the sixteen conventional colours `black`, `silver` etc, or a RGB hexadecimal color specification of the form `#`*XXXXXX*. Note that the argument *color* is processed, as a consequence numerical color arguments should be given as `\#`*XXXXXX*.

`\@nostyle` Close active text-level declarations and ignore further text-level declarations. The effect stops when the enclosing block-level element is closed.

`\@clearstyle` Simply close active text-level declarations.

Notice on font styling with CSS

The preferred way to style text in new versions of the HTML "standard" is using style-sheet specifications. Those can be given as argument to a "`style`" attributes of HTML elements, most noticeably of the `span` elements. For instance, to get italics in old versions of HTML one used the text-level "`i`" element as in `<i>...</i>`. Now, for the same results of getting italics one may write: `<span style="font-style:italic">...</span>`. An indeed `hevea` styles text in that manner, starting from version 2.00. Such (verbose) declarations are then abstracted into style class declarations by HeVeA optimiser `esponja`, which is invoked by `hevea` when given option "`-O`".

Notice that style attributes can be given to elements other than `span`. However, combining style attributes requires a little care as only one style attribute is allowed. Namely `<cite style="font-weight:bold" style="color:red">` is illegal and should be written `<cite style="font-weight:bold;color:red">`.

The command `\@addtyle` can be handy for adding style to already style elements:

`\@addstyle{`*name:val*`}{`*attrs*`}` Echo the space-separated attributes *attrs* of a tag with the *name:val* style declaration added to these attributes. The `style` attribute is added if necessary. Examples: `\@addstyle{color:red}{href` will produce `href="#" style="color:red"`, and `\@addstyle{color:red}{href="#" style="font-style:italic"}`

will produce `href="#"  style="font-style:italic;color:red"`. Note that an unnecessary extra space can be added in some cases.

As an example, consider the following definition of a command for typesetting citation in bold, written directly in HTML:

```
\newcommand{\styledcite}[2][]
{{\@styleattr{cite}{\@addstyle{#1}{style="font-weight:bold"}}#2}}
```

The purpose of the optional argument is to add style to specific citations, as in:

```
Two fundamental works: \styledcite{The Holy Bible} and
\styledcite[color:red]{Das Kapital}.
```

Notice that the example is given for illustrating the usage of the `\@addstyle` macros, which is intended for package writers. A probably simpler way to proceed would be to use LaTeX text-style declarations:

```
\newcommand{styledcite}[2][]{{\@style{cite}#1\bf{}#2}}
Two fundamental works: \styledcite{The Holy Bible} and
\styledcite[\color{red}]{Das Kapital}.
```

8.4 The `rawhtml` environment

Any text enclosed between `\begin{rawhtml}` and `\end{rawhtml}` is echoed verbatim into the HTML output file. Similarly, `\rawhtmlinput{`*file*`}` echoes the contents of file *file*. In fact, `rawhtml` is the environment counterpart of the `\@print` command, but experience showed it to be much more error prone.

When HeVeA was less sophisticated then it is now, `rawhtml` was quite convenient. But, as time went by, numerous pitfalls around `rawhtml` showed up. Here are a few:

- Verbatim means that no translation of any kind is performed. In particular, be aware that input encoding (see B.17.4) does not apply. Hence one should use ascii only, if needed non-ascii characters can be given as entity or numerical character references — *e.g.* `&eacute;` or `&#XE9;` for é.
- The `rawhtml` environment should contain only HTML text that makes sense alone. For instance, writing `\begin{rawhtml}<table>\end{rawhtml}`... `\begin{rawhtml}</table>\end{rawhtml}` is dangerous, because HeVeA is not informed about opening and closing the block-level element `table`. In that case, one should use the internal macros `\@open` and `\@close`.
- `\begin{rawhtml}`*text*`\end{rawhtml}` fragments that contain block-level elements will almost certainly mix poorly with `p` elements (introduced by paragraph breaks) and with active style declaration (introduced by, for instance, `\it`). Safe usage will most of the time means using the internal macros `\@nostyle` and `\@out@par`.
- When HeVeA is given the command-line option `-O`, checking and optimisation of text-level elements in the whole document takes place. As a consequence, incorrect HTML introduced by using the `rawhtml` environment may be detected at a later stage, but this is far from being certain.

As a conclusion, do not use the `rawhtml` environment! A much safer option is to use the `htmlonly` environment and to write LaTeX code. For instance, in place of writing:

```
\begin{rawhtml}
A list of links:
<ul>
<li><a href="http://www.apple.com/">Apple</a>.
<li><a href="http://www.sun.com/">Sun</a>.
</ul>
\end{rawhtml}
```

One can write:

```
\begin{htmlonly}
A list of links:
\begin{itemize}
\item \ahref{http://www.apple.com/}{Apple}.
\item \ahref{http://www.sun.com/}{Sun}.
\end{itemize}
\end{htmlonly}
```

If HeVeA is targeted to text or info files (see Section 11). The text inside `rawhtml` environments is ignored. However there exists a `rawtext` environment (and a `\rawtextinput` command) to echo text verbatim in text or info output mode. Additionally, the `raw` environment and a `\rawinput` command echo their contents verbatim, regardless of HeVeA output mode. Of course, when HeVeA produces HTML, the latter environment and command suffer from the same drawbacks as `rawhtml`.

8.5 Examples

As a first example of using internal macros, consider the following excerpt from the `hevea.hva` file that defines the `center` environment:

```
\newenvironment{center}{\@open{div}{style="text-align:center"}}{\@close{div}}
```

Notice that the code above is no longer present and is given here for explanatory purpose only. Now HeVeA uses style-sheets and the actual definition of the `center` environment is as follows:

```
\newstyle{.center}{text-align:center;margin-left:auto;margin-right:auto;}%
\setenvclass{center}{center}%
\newenvironment{center}
  {\@open{div}{\@getprint{class="\getenvclass{center}"}}
  {\@close{div}}%
```

Basically environments `\begin{center}`... `\end{center}` will, by default, be translated into blocks `<div class="center">`.. Additionally, the style class associated to `center` environments is managed through an indirection, using the commands `\setenvclass` and `\getenvclass`. See section 9.3 for more explanations.

Another example is the definition of the `\purple` color declaration (see section 8.1.2):

```
\newcommand{\purple}{\@fontcolor{purple}}
```

HeVeA does not feature all text-level elements by default. However one can easily use them with internal macros. For instance this is how you can make all emphasised text blink:

```
\renewcommand{\em}{\@styleattr{em}{style="text-decoration:blink"}}
```

Then, here is the definition of a simplified `\imgsrc` command (see section 8.1.1), without its optional argument:

```
\newcommand{\imgsrc}[1]
  {\@print{<img src="}\@getprint{#1}\@print{">}}
```

Here, `\@print` and `\@getprint` are used to output HTML text, depending upon whether this text requires processing or not. Note that `\@open{img}{src="#1"}` is not correct, because the element `img` consists in a single tag, without a closing tag.

Another interesting example is the definition of the command `\@doaelement`, which HeVeA uses internally to output `A` elements.

```
\newcommand{\@doaelement}[2]
  {{\@nostyle\@print{<a }\@getprint{#1}\@print{>}}{#2}{\@nostyle\@print{</a>}}
```

The command `\@doaelement` takes two arguments: the first argument contains the opening tag attributes; while the second element is the textual content of the `A` element. By contrast with the `\imgsrc` example above, tags are emitted inside groups where styles are cancelled by using the `\@nostyle` declaration. Such a complication is needed, so as to avoid breaking proper nesting of text-level elements.

Here is another example of direct block opening. The `bgcolor` environment from the `color` package locally changes background color (see section B.14.2.1). This environment is defined as follows:

```
\newenvironment{bgcolor}[2][style="padding:1em"]
{\@open{table}{}\@open{tr}{}%
\@open{td}{\@addstyle{background-color:\@getcolor{#2}}{#1}}}
{\@close{td}\@close{tr}\@close{table}}
```

The `bgcolor` environment operates by opening a HTML table (`table`) with only one row (`tr`) and cell (`td`) in its opening command, and closing all these elements in its closing command. In my opinion, such a style of opening block-level elements in environment opening commands and closing them in environment closing commands is good style. The one cell background color is forced with a `background-color` property in a `style` attribute. Note that the mandatory argument to `\begin{bgcolor}` is the background color expressed as a high-level color, which therefore needs to be translated into a low-level color by using the `\@getcolor` internal macro from the `color` package. Additionally, `\begin{bgcolor}` takes HTML attributes as an optional argument. These attributes are the ones of the `table` element.

If you wish to output a given Unicode character whose value you know, the recommended technique is to define an ad-hoc command that simply call the `\@print@u` command. For instance, "blackboard sigma" is Unicode `U+02140` (hexa). Hence you can define the command `\bbsigma` as follows:

```
\newcommand{\bbsigma}{\@print@u{X2140}}
```

8.6 The document charset

According to standards, as far as I understand them, HTML pages are made of Unicode (ISO 10646) characters. By contrast, a file in any operating system is usually considered as being made of bytes.

To account for that fact, HTML pages usually specify a *document charset* that defines a translation from a flow of bytes to a flow of characters. For instance, the byte `0xA4` means Unicode `0x00A4` (¤) in the ISO-8859-1 (or latin1) encoding, and `0x20AC` (€) in the ISO-8859-15 (or latin9) encoding. Notice that HeVeA has no difficulty to output both symbols, in fact they are defined as Unicode characters:

```
\newcommand{\textcurrency}{\@print@u{XA4}}
\newcommand{\texteuro}{\@print@u{X20AC}}
```

But the `\@print@u` command may output the specified character as a byte, when possible, by the means of the *output translator*. If not possible, `\@print@u` outputs a numerical character references (for instance `&#X20AC;`).

Of course, the document charset and the output translator must be synchronised. The command `\@def@charset` takes a charset name as argument and performs the operation of specifying the document character set and the output translator. It should occur in the document preamble. Valid charset names are `ISO-8859-`n where n is a number in `1`...`15`, `KOI8-R`, `US-ASCII` (the default), `windows-`n where n is `1250`, `1251`, `1252` or `1257`, or `macintosh`, or `UTF-8`. In case those charsets do not suffice, you may ask the author for other document charsets. Notice however that document charset is not that important, the default `US-ASCII` works everywhere! *Input* encoding of source files is another, although related, issue — see Section B.17.4.

If wished so, the charset can be extracted from the current locale environment, provided this yields a valid (to HeVeA) charset name. This operation is performed by a companion script: `xxcharset.exe`. It thus suffices to launch HeVeA as:

`# hevea -exec xxcharset.exe` *other arguments*

9 Support for style sheets

9.1 Overview

Starting with version 1.08, HeVeA offers support for style sheets (of the CSS variant see [CSS-2]).

Style sheets provide enhanced expressiveness. For instance, it is now possible to get "real" (whatever real means here) small caps in HTML, and in a relatively standard manner. There are other, discrete, maybe unnoticeable, similar enhancements.

However, style sheets mostly offer an additional mechanism to customise their documents to HeVeA users. To do so, users should probably get familiar with how HeVeA uses style sheets in the first place.

HeVeA interest for style sheets is at the moment confined to block-level elements (`div`, `table`, `H<`*n*`>`, etc.). The general principle is as follows: when a command or an environment gets translated into a block-level element, the opening tag of the block level element has a `class="`*name*`"` attribute, where *name* is the command or environment name.

As an example the LaTeX command `\subsection` is implemented with the element `h3`, resulting in HTML output of the form:

```
<h3 class="subsection">
...
</h3>
```

By default, most styles are undefined, and default rendering of block-level elements applies. However, some packages (such as, for instance `fancysection`, see Section B.16.4) may define them. If you wish to change the style of section headers, loading the `fancysection` package may prove appropriate (see B.16.4). However, one can also proceed more directly, by appending new definitions to the *document style sheet*, with the command `\newstyle`. For instance, here is a `\newstyle` to add style for subsections.

```
\newstyle{.subsection}{padding:1ex;color:navy;border:solid navy;}
```

This declaration adds some style element to the `subsection` class (notice the dot!): blocks that declare to belong to the class will show dark-blue text, some padding (space inside the box) is added and a border will be drawn around the block. These specification will normally affect all subsections in the document.

The following points are worth noticing:

- To yield some effect, `\newstyle` commands *must* appear in the document preamble, *i.e.* before `\begin{document}`.
- Arguments to `\newstyle` commands are processed.
- The `hevea` package defines all style sheet related commands as no-ops. Thus, these commands do not affect document processing by LaTeX.

9.2 Changing the style of all instances of an environment

In this very document, all `verbatim` environments appear over a light green background, with small left and right margins. This has been performed by simply issuing the following command in the document preamble.

```
\newstyle{.verbatim}{margin:1ex 1ex;padding:1ex;background:\#ccffcc;}
```

Observe that, in the explicit numerical color argument above, the hash character "`#`" has to be escaped.

9.3 Changing the style of some instances of an environment

One can also change the style class attached to a given instance of an environment and thus control styling of environments more precisely.

As a matter of fact, the name of the class attribute of environment *env* is referred to through an indirection, by using the command `\getenvclass{`*env*`}`. The class attribute can be changed with the command `\setenvclass{`*env*`}{`*class*`}`. The `\setenvclass` command internally defines a command `\`*env*`@class`, whose content is read by the `\getenvclass` command. As a consequence, the class attribute of environments follows normal scoping rules. For instance, here is how to change the style of *one* `verbatim` environment.

```
{\setenvclass{verbatim}{myverbatim}
\begin{verbatim}
This will be styled through class 'myverbatim', introduced by:
\newstyle{.myverbatim}
  {margin:1ex 3x;padding:1ex;
   color:maroon;
   background:\@getstylecolor[named]{Apricot}}
\end{verbatim}}
```

Observe how the class of environment `verbatim` is changed from its default value to the new value `myverbatim`. The change remains active until the end of the current group (here, the "`}`" at the end). Then, the class of environment `verbatim` is restored to its default value — which happen to be `verbatim`.

This example also shows two new ways to specify colours in style definition, with a conventional HTML color name (here maroon) or as a high-level color (see Section B.14.2), given as an argument to the `\@getstylecolor` internal command (here `Apricot` from the `named` color model).

A good way of specifying style class changes probably is by defining new environments.

```
\newenvironment{flashyverbatim}
  {\setenvclass{verbatim}{myverbatim}\verbatim}
  {\endverbatim}
```

Then, we can use `\begin{flashyverbatim}`... `\end{flashyverbatim}` to get verbatim environments style with the intended `myverbatim` style class.

9.4 Which class affects what

Generally, the styling of environment *env* is performed through the commands `\getenvclass{`*env*`}` and `\setenvclass{`*env*`}{`...`}`, with `\getenvclass{`*env*`}` producing the default value of *env*.

Concretely, this means that most of the environments are styled through an homonymous style class. Here is a non-exhaustive list of such environments

> figure, table, itemize, enumerate, list, description, trivlist, center, flushleft, flushright, quote, quotation, verbatim, abstract, mathpar (cf Section B.17.15), lstlisting (cf. Section B.17.13), etc.

All sectioning commands (`\part`, `\section` etc.) output `H<`*n*`>` block-level elements, which are styled through style classes named `part`, `section`, etc.

List making-environment introduce extra style classes for items. More specifically, for list-making environments `itemize` and `enumerate`, `li` elements are styled as follows:

```
<ul class="itemize">
<li class="li-itemize"> ...
</ul>
```

```
<ol class="enumerate">
<li class="li-enumerate"> ...
</ol>
```

That is, `li` elements are styled as environments, the key name being `li-`*env*.

The `description`, `trivlist` and `list` environments (which all get translated into `DL` elements) are styled in a similar way, internal `DT` and `DD` elements being styles through names `dt-`*env* and `dd-`*env* respectively.

9.5 A few examples

9.5.1 The title of the document

The command `\maketitle` formats the document title within a `table` element, with class `title`, for display. The name of the title is displayed inside block `h1`, with class `titlemain`, while all other information (author, date) are displayed inside block `h3`, with class `titlerest`.

```
<table class="title">
 <tr>
  <td style="padding:1ex">
   <h1 class="titlemain">..title here..</h1>
   <h3 class="titlerest">..author here..</h3>
   <h3 class="titlerest">..date here..</h3>
  </td>
 </tr>
</table>
```

Users can impact on title formatting by adding style in the appropriate style classes. For instance the following style class definitions:

```
\newstyle{.title}
  {text-align:center;margin:1ex auto;color:navy;border:solid navy;}
\newstyle{.titlerest}{font-variant:small-caps;}
```

will normally produce a title in dark blue, centered in a box, with author and date in small-caps.

9.5.2 Enclosing things in a styled `div`

At the moment, due to the complexity of the task, environments `tabular` and `array` cannot be styled as others environments can be, by defining an appropriate class in the preamble. However, even for such constructs, limited styling can be performed, by using the `divstyle` environment. The opening command `\begin{divstyle}{`*class*`}` takes the name of a class as an argument, and translates to `<div class="`*class*`">`. Of course the closing command `\end{divstyle}` translates to `</div>`. The limitation is that the enclosed part may generate more HTML blocks, and that not all style attribute defined in class class *class* will apply to those inner blocks.

As an example consider the style class definition below.

```
\newstyle{.ruled}{border:solid black;padding:1ex;background:\#eeddbb;color:maroon}
```

The intended behaviour is to add a black border around the inner block (with some padding), and to have text over a light brown background.

If we, for instance, enclose an `itemize` environment, the resulting effect is more or less what we have expected:

```
\begin{divstyle}{ruled}
\begin{itemize}
\item A ruled itemize
\item With two items.
\end{itemize}
\end{divstyle}
```

However, enclosing a centered `tabular` environment in a `divstyle{ruled}` one is less satisfactory.

```
\begin{divstyle}{ruled}
\begin{center}\begin{tabular}{|c|c|}
\hline \bf English & \bf French\\ \hline
Good Morning & Bonjour\\ Thank You & Merci\\ Good Bye & Au Revoir\\ \hline
\end{tabular}\end{center}
\end{divstyle}
```

In the HTML version of this document, one sees that the brown background extend on all the width of the displayed page.

This problem can be solved by introducing an extra table. We first open an extra centered table and then only open the `divstyle` environment.

```
\begin{center}\begin{tabular}{c}
\begin{divstyle}{ruled}
\begin{tabular}{|c|c|}
\hline \bf English & \bf French\\ \hline
Good Morning & Bonjour\\ Thank You & Merci\\ Good Bye & Au Revoir\\
\hline
\end{tabular}
\end{divstyle}
\end{tabular}\end{center}
```

This works because of the rules that govern the width of HTML `table` elements, which yield minimal width. This trick is used in numerous places by HEVEA, for instance in document titles, and looks quite safe.

Another solution is to specify the `display` property of the styling `div` block as being `inline-block`:

```
\newstyle{.ruledbis}
  {border:solid black;padding:1ex;background:\#eeddbb;color:maroon;display:inline-block;}
```

9.5.3 Styling the `itemize` environment

Our idea is highlight lists with a left border whose color fades while lists are nested. Such a design may be appropriate for tables of content, as the one of this document. The text above is typeset from the following LaTeX source.

```
\begin{toc}
\item Part~A
\begin{toc}
\item Chapter~I
\begin{toc}
\item Section~I.1
\item Section~I.2
\end{toc}
  ...
\end{toc}
\end{toc}
```

For simplicity, we assume a limit of four over the nesting depth of `toc` environment. We first define four style classes `toc1`, `toc2`, `toc3` and `toc4` in the document preamble. Since those classes are similar, a command `\newtocstyle` is designed.

```
\newcommand{\newtocstyle}[2]
{\newstyle{.toc#1}{list-style:none;border-left:1ex solid #2;padding:0ex 1ex;}}
```

```
\newtocstyle{1}{\@getstylecolor{Sepia}}
\newtocstyle{2}{\@getstylecolor{Brown}}
\newtocstyle{3}{\@getstylecolor{Tan}}
\newtocstyle{4}{\@getstylecolor{Melon}}
```

The `toc` environment uses a counter to record nesting depth. Notice how the style class of the `itemize` environment is redefined before `\begin{itemize}`.

```
\newcounter{toc}
\newenvironment{toc}
{\stepcounter{toc}\setenvclass{itemize}{toc\thetoc}\begin{itemize}}
{\addtocounter{toc}{-1}\end{itemize}}
```

The outputted HTML is:

```
<ul class="toc1"><li class="li-itemize">
Part A
<ul class="toc2"><li class="li-itemize">
Chapter I
<ul class="toc3"><li class="li-itemize">
Section I.1
<li class="li-itemize">Section I.2
  ...
</ul>
</ul>
```

9.6 Miscellaneous

9.6.1 HaCha and style sheets

HaCha now produces an additional file: a style sheet, which is shared by all the HTML files produced by HaCha. Please refer to section 7.1 for details.

9.6.2 Producing an external style sheet

By default, style declarations defined with `\newstyle` go into the header of the HTML document *doc*`.html`. However, one can send those declaration into an external style file, whose name is *doc*`.css`. Then, HeVeA automatically relates *doc*`.html` to its style sheet *doc*`.css`. To achieve this behaviour, it suffices to set the value of the boolean register `externalcss` to *true*, by issuing the command `\externalcsstrue` in the preamble of the source document. Notice that HeVeA output still can be processed by HaCha, with correct behaviour.

9.6.3 Linking to external style sheets

The HeVeA command `\loadcssfile{`*url*`}` allows the user to link to an external style sheet (like the link option for HTML). The command takes an *url* of the external sheet as argument and emits the HTML text to *link* to the given external style sheet. As an example, the command

```
\loadcssfile{../abc.css}
```

produces the following HTML text in the `head` of the document.

```
<link rel="stylesheet" type="text/css" href="../abc.css">
```

To yield some effect, `\loadcssfile` must appear in the document preamble. Several `\loadcssfile` commands can be issued. Then the given external style sheets appear in the output, following source order.

Notice that the argument to `\loadcssfile` is processed. Thus, if it contains special characters such as "#" or "$", those must be specified as `\#` and `\$` respectively. A viable alternative would be to quote the argument using the `\url` command from the `url` package (see Section B.17.11).

9.6.4 Limitations

At the moment, style class definitions cumulate, and appear in the `style` element in the order they are given in the document source. There is no way to cancel the default class definitions performed by HeVeA before it starts to process the user's document. Additionally, external style sheets specified with `\loadcssfile` appear before style classes defined with `\newstyle`. As a consequence (if I am right), styles declared by `\newstyle` take precedence over those contained in external style sheets. Thus, using external style-sheets, especially if they alter the styling of elements, may produce awkward results.

Those limitations do not apply of course to style classes whose names are new, since there cannot be default definitions for them. Then, linking with external style sheets can prove useful to promote uniform styling of several documents produced by HeVeA.

10 Customising HeVeA

HeVeA can be controlled by writing LaTeX code. In this section, we examine how users can change HeVeA default behaviour or add functionalities. In all this section we assume that a document `doc.tex` is processed, using a private command file `macros.hva`. That is, HeVeA is invoked as:

```
# hevea macros.hva doc.tex
```

The general idea is as follows: one redefines LaTeX constructs in `macros.hva`, using internal commands. This requires a good working knowledge of both LaTeX and HTML. Usually, one can avoid internal commands, but then, all command redefinitions interact, sometimes in very nasty ways.

10.1 Simple changes

Users can easily change the rendering of some constructs. For instance, assume that *all* quotations in a text should be emphasised. Then, it suffices to put the following re-declaration in `macros.hva`:

```
\renewenvironment{quote}
  {\@open{blockquote}{}\@style{em}}
  {\@close{blockquote}}
```

The same effect can be achieved without using any of the internal commands:

```
\let\oldquote\quote
\let\oldendquote\endquote
\renewenvironment{quote}{\oldquote\em}{\oldendquote}
```

In some sense, this second solution is easier, when one already knows how to customise LaTeX. However, this is less safe, since the definition of `\em` can be changed elsewhere.

There is yet another solution that takes advantage of style sheets. One can also add this line to the `macros.hva` file:

```
\newstyle{.quote}{font-style:oblique;}
```

This works because the environment `quote` is styled through style class `quote` (see Section 9.2). Notice that this solution has very little to do with "*emphasising*" in the proper sense, since here we short-circuit the implicit path from `\em` to oblique fonts.

10.2 Changing defaults for type-styles

HeVeA default rendering of type style changes is described in section B.15.1. For instance, the following example shows the default rendering for the font shapes:

```
\itshape italic shape \slshape slanted shape
\scshape small caps shape \upshape upright shape
```

By default, `\itshape` is italics, `\slshape` is oblique italics, `\scshape` is small-caps (thanks to style sheets) and `\upshape` is no style at all. All shapes are mutually exclusive, this means that each shape declaration cancels the effect of other active shape declarations. For instance, in the example, small caps shapes is small caps (no italics here).

If one wishes to change the rendering of some of the shapes (say slanted caps), then one should redefine the old-style `\sl` declaration. For instance, to render slanted as Helvetica (why so?), one should redefine `\sl` by `\renewcommand{\sl}{\@span{style="font-family:Helvetica"}}` in `macros.hva`.

Hence, redefining old-style declarations using internal commands should yield satisfactory output. However, since cancellation is done at the HTML level, a declaration belonging to one component may sometimes cancel the effect of another that belongs to another component. Anyway, you might have not noticed it if I had not told you.

10.3 Changing the interface of a command

Assume for instance that the base style of `doc.tex` is *jsc* (the *Journal of Symbolic Computation* style for articles). For running HeVeA, the *jsc* style can be replaced by *article* style, but for a few commands whose calling interface is changed. In particular, the `\title` command takes an extra optional argument (which HeVeA should ignore anyway). However, HeVeA can process the document as it stands. One solution to insert the following lines into `macros.hva`:

```
\input{article.hva}% Force document class 'article'
\let\oldtitle=\title
\renewcommand{\title}[2][]{\oldtitle{#2}}
```

The effect is to replace `\title` by a new command which calls HeVeA `\title` with the appropriate argument.

10.4 Checking the optional argument within a command

HeVeA fully implements LaTeX 2$_\varepsilon$ `\newcommand`. That is, users can define commands with an optional argument. Such a feature permits to write a `\epsfbox` command that has the same interface as the LaTeX command and echoes itself as it is invoked to the *image* file. To do this, the HeVeA `\epsfbox` command has to check whether it is invoked with an optional argument or not. This can be achieved as follows:

```
\newcommand{\epsfbox}[2][!*!]{%
\ifthenelse{\equal{#1}{!*!}}
{\begin{toimage}\epsfbox{#2}\end{toimage}}%No optional argument
{\begin{toimage}\epsfbox[#1]{#2}\end{toimage}}}%With optional argument
\imageflush}
```

10.5 Changing the format of images

Semi-automatic generation of included images is described in section 6. Links to included images are generated by the `\imageflush` command, which calls the `\imgsrc` command:

```
\newcommand{\imageflush}[1][]
{\@imageflush\stepcounter{image}\imgsrc[#1]{\hevaimagedir\jobname\theimage\heveaimageext}}
```

That is, you may supply a HTML-style attribute to the included image, as an optional argument to the `\imageflush` command.

By default, images are PNG images stored in `.png` files. HeVeA provides support for the alternative GIF image file format. It suffices to invoke `hevea` as:

```
# hevea gif.hva doc.tex
```

Then `imagen` must be run with option `-gif`:

```
# imagen -gif doc
```

A convenient alternative is to invoke `hevea` as:

```
# hevea -fix gif.hva doc.tex
```

Then `hevea` will invoke `imagen` with the appropriate option when it thinks images need to be rebuild. An even more convenient alternative is to load `gif.hva` from within document source, for instance with the `\usepackage` command.

HEVEA also provides support for the alternative SVG image file format. As for GIF images, it is more convenient to use option `-fix` to combine `hevea` and `imagen` invocations:

```
# hevea -fix svg.hva doc.tex
```

Notice that `imagen` production chain of SVG images always call `pdflatex`, even when *not* given the `-pdf` command-line option. Hence the source code of images must be processable by `pdflatex`. This precludes using `latex`-only packages such as pstricks for instance.

As not all browsers display SVG images, `hevea` and `imagen` are bit special: `imagen` produces both PNG[2] and SVG images; while `hevea` offers both image sources, letting client browser select the most appropriate one by the means of teh `srcset` attribute of the `img` element.

10.6 Storing images in a separate directory

By redefining the `\heveaimagedir` command, users can specify a directory for images. More precisely, if the following redefinition occurs in the document preamble.

```
\renewcommand{\heveaimagedir}{dir}
```

Then, all links to images in the produced HTML file will be as "*dir*/...". Then `imagen` must be invoked with option `- todir`:

```
# imagen -todir dir doc
```

As usual, `hevea` will invoke `imagen` with the appropriate option, provided it is passed the `-fix` option.

10.7 Controlling `imagen` from document source

The internal command `\@addimagenopt{`*option*`}` add the text *option* to `imagen` command-line options, when launched automatically by `hevea` (*i.e.* when `hevea` is given the `-fix` command-line option).

For instance, to instruct `hevea`/`imagen` to reduce all images by a factor of $\sqrt{2}$, it suffices to state:

```
%HEVEA\@addimagenopt{-mag 707}
```

See section C.1.5 for the list of command-line options accepted by `imagen`.

11 Other output formats

It is possible to translate LaTeX file into other formats than HTML. There are two such formats: plain text and info files. This enables producing postscript, HTML, plain text and info manuals from one (LaTeX) input file.

[2] or GIF, if `gif.hva` is loaded

11.1 Text

The LaTeX file is processed and converted into a plain text formatted file. It allows some pretty-printing in plain text.

To translate into text, invoke HeVeA as follow:

```
# hevea -text [-w <width>] myfile.tex
```

Then, HeVeA produces `myfiles.txt` a plain text translation of `myfile.tex`.

Additionally, the optional argument `-w <number>` sets the width of the output for text formatting. By default, The text will be 72 characters wide.

Nearly every environment has been translated, included lists and tables. The support is nearly the same as in HTML, excepted in some cases described hereafter.

Most style changes are ignored, because it is hardly possible to render them in plain text. Thus, there are no italics, bold fonts, underlinings, nor size change or colours... The only exception is for the verbatim environment that puts the text inside quotes, to distinguish it more easily.

Tables with borders are rendered in the same spirit as in LaTeX. Thus for instance, it is possible to get vertical lines between some columns only. Table rendering can be poor in case of line overflow. The only way to correct this (apart from changing the tables themselves) is to adjust the formatting width, using the the `-w` command-line option.

For now, maths are not supported at all in text mode. You can get very weird results with in-text mathematical formulas. Of course, simple expressions such as subscripts remains readable. For instance, x^2 will be rendered as `x^2`, but $\int_0^1 f(x)dx$ will yield something like : `int01f(x)dx`.

11.2 Info

The file format info is also supported. Info files are text files with limited hypertext links, they can be read by using *emacs* info mode or the `info` program. Please note that HeVeA translates plain LaTeX to info, and not TeXinfo.

You can translate your LaTeX files into info file(s) as follows:

```
# hevea -info [-w <width>] myfile.tex
```

Then, HeVeA produces the file `myfile.info`, an info translation of `myfile.tex`. However, if the resulting file is too large, it is cut into pieces automatically, and `myinfo.info` now contains references for all the nodes in the others files, which are named `myfile.info-1`, `myfile.info-2`,...

The optional argument `-w` has the same meaning as for text output.

The text will be organised in nodes that follow the pattern of LaTeX sectioning commands. Menus are created to navigate through the sections easily

A table of content is produced automatically. References, indexes and footnotes are supported, as they are in HTML mode. However, the info format only allows pointers to info nodes, *i.e.* in HeVeA case, to sectional units. As a consequence all cross references lead to sectional unit headers.

Part B
Reference manual

This part follows the pattern of the LaTeX reference manual [LaTeX, Appendix C].

B.1 Commands and Environments

B.1.1 Command Names and Arguments

LaTeX comments that start with "`%`" and end at end of line are ignored and produce no output. Usually, HeVeA ignore such comments. However, HeVeA processes text that follows "`%HEVEA`" and some other comments have a specific meaning to it (see section 5.3).

Command names follow strict LaTeX syntax. That is, apart from `#`, `$`, `~`, `_` and `^`, they either are "\" followed by a single non-letter character or "\" followed by a sequence of letters. Additionally, the letter sequence may be preceded by "`@`" (and this is the case of many of HeVeA internal commands), or terminated by "`*`" (starred variants are implemented as plain commands).

Users are strongly advised to follow strict LaTeX syntax for arguments. That is, mandatory arguments are enclosed in curly braces `{... }` and braces inside arguments must be properly balanced. Optional arguments are enclosed in square brackets `[... ]`. However, HeVeA does its best to read arguments even when they are not enclosed in curly braces. Such arguments are a single, different from "\", "`{`" and " ", character or a command name. Thus, constructs such as `\'ecole`, `$a_1$` or `$a_\Gamma$` are recognized and processed as école a_1 and a_Γ. By contrast, `a^\mbox{...}` is not recognized and must be written `a^{\mbox{...}}`.

Also note that, by contrast with LaTeX, comments are parsed during argument scanning, as an important consequence brace nesting is also checked inside comments.

With respect to previous versions, HeVeA has been improved as regards emulation of complicated argument passing. That is, commands and their arguments can now appear in different static text bodies. As a consequence, HeVeA correctly processes the following source:

```
\newcommand{\boite}{\textbf}
\boite{In bold}
```

The definition of `\boite` makes it reduces as `\textbf` and HeVeA succeeds in fetching the argument "`{In bold}`". We get

---

In bold

The above example arguably is no "legal" LaTeX, but HeVeA handles it. Of course, there remains numerous "clever" LaTeX tricks that exploits TeX internal behaviour, which HeVeA does not handle. For instance consider the following source:

```
\newcommand{\boite}[1]{\textbf#1}
\boite{{In bold}, Not in Bold.}
```

LaTeX typesets the text "In bold" using bold font, leaving the rest of the text alone. While HeVeA typesets everything using bold font. Here is LaTeX output:

In bold, Not in Bold.

Note that, in most similar situations, HeVeA will likely crash.

As a conclusion of this important section, Users are strongly advised to use ordinary command names and curly braces and not to think too much the TeX way.

B.1.2 Environments

Environment opening and closing is performed like in LaTeX, with `\begin{`*env*`}` and `\end{`*env*`}`. The `*`-form of an environment is a plain environment.

It is not advised to use `\`*env* and `\end`*env* in place of `\begin{`*env*`}` and `\end{`*env*`}`.

B.1.3 Fragile Commands

Fragile commands are not relevant to HeVeA and `\protect` is defined as a null command.

B.1.4 Declarations

Scope rules are the same as in LaTeX.

B.1.5 Invisible Commands

I am a bit lost here. However spaces in the output should correspond to users expectations. Note that, to HeVeA being invisible commands is a static property attached to command name.

B.1.6 The \\ Command

The \\ and * commands are the same, they perform a line break, except inside arrays where they end the current row. Optional arguments to \\ and * are ignored.

B.2 The Structure of the Document

Document structure is a bit simplified with respect to LaTeX, since documents consist of only two parts. The *preamble* starts as soon as HeVeA starts to operate and ends with the `\begin{document}` construct. Then, any input occurring before `\end{document}` is translated to HTML. However, the preamble is processed and the preamble comprises the content of the files given as command-line arguments to HeVeA, see section C.1.1.1). As a consequence, command and environment definitions that occur before `\begin{document}` are performed. and they remain valid during all the processing.

In particular one can define a *header* and a *footer*, by using the `\htmlhead` and `\htmlfoot` commands in the preamble. Those commands register their argument as the header and the footer of the final HTML document. The header appears first while the footer appears last in (visible) HTML output. This is mostly useful when HeVeA output is later cut into pieces by HaCha, since both header and footer are replicated at the start and end of any file generated by HaCha. For instance, to append a copyright notice at the end of all the HTML pages, it suffices to invoke the `\htmlfoot` command as follows in the document preamble:

```
\htmlfoot{\copyright to me}
```

The `\htmlhead` command cannot be used for changing anything outside of the HTML document body, there are specific commands for doing this. Those command must be used in the document preamble. One can change HeVeA default (empty) attribute of the opening `<body ...>` tag by redefining `\@bodyargs`. For instance, you get black text on a white background, when the following declaration occurs before `\begin{document}`:

```
\renewcommand{\@bodyargs}{style="color:black;background:white"}
```

Since version 1.08, a recommended alternative is to use style sheets:

```
\newstyle{body}{color:black; background:white;}
```

One can also change the default (empty) attribute of the opening `<html ...>` tag by redefining `\@htmlargs`. For instance you can set the language attribute of the whole document by issuing the following redefinition in the document preamble:

```
\renewcommand{\@htmlargs}{lang=en}
```

Similarly, some elements can be inserted into the output file `head` element by redefining the `\@meta` command (Such elements typically are `meta`, `link`, etc.). As such text is pure HTML, it should be included in a `rawhtml` environment. For instance, you can specify author information as follows:

```
\let\oldmeta=\@meta
\renewcommand{\@meta}{%
\oldmeta
\begin{rawhtml}
<meta name="Author" content="Luc Maranget">
\end{rawhtml}}
```

Note how `\@meta` is first bound to `\oldmeta` before being redefined and how `\oldmeta` is invoked in the new definition of `\@meta`. Namely, simply overriding the old definition of `\@meta` would imply not outputting default meta-information.

The `\@charset` command holds the value of the (HTML) document character set. By default, this value is `US-ASCII`. In previous versions of HeVeA, one could change the value of the document character set by simply redefining `\@charset`. Then, it was users responsability to provide a (LaTeX) document in the correspounding encoding. This is no longer so, and users should *not* redefine `\@charset` directly. Please, see Section 8.6 for details.

B.3 Sentences and Paragraphs

B.3.1 Spacing

Generally speaking, spaces (and single newline characters) in the source are echoed in the output. Browser then manage with spaces and line-breaks. Following LaTeX behaviour, spaces after commands are not echoed. Spaces after invisible commands with arguments are not echoed either.

However this is no longer true in math mode, see section B.7.7 on spaces in math mode.

B.3.2 Paragraphs

New paragraphs are introduced by one blank line or more. Paragraphs are not indented. Thus the macros `\indent` and `\noindent` perform no action. Paragraph are rendered by `p` elements. In some occasions, this technique may produce spurious paragraphs (see 3.1.1).

B.3.3 Footnotes

The commands `\footnote`, `\footnotetext` and `\footnotemark` (with or without optional arguments) are supported. The `footnote` counter exists and (re)setting it or redefining `\thefootnote` should work properly. When footnotes are issued by a combination of `\footnotemark` and `\footnotetext`, a `\footnotemark` command must be issued first, otherwise some footnotes may get numbered incorrectly or disappear. Footnotes appear at document end in the *article* style and at chapters end in the *book* style. See section 7.3.7 for a description of how footnotes are flushed.

B.3.4 Accents and special symbols

Thanks to Unicode character references, HeVeA can virtually output any symbol. It may happen that HeVeA does not known about a particular symbol, that is, most of the time, HeVeA does not known about a particular command. In that case a warning is issued to draw user attention. Users can then choose a particular symbol amongst the recognized ones, or as an explicit Unicode character reference (see Section 4.2 for an example of this technique).

Commands for making accents used in non-English languages, such as `\'`, work when applied to accent-less (*i.e.* ascii) letters and that the corresponding accented letters exist in the Unicode character set. Otherwise, the argument to the command is not modified and a warning is issued. For instance, consider the following source code, where, after a legitimate use of acute accents, one attempt to put an accute accent over the letter “h”:

```
``\'Ecole'' works as in \LaTeX, while ``\'h'' does not.
```

HeVeA output will be "École" works as in LaTeX, while "h" does not. And a warning will be issued.

```
./tmp.tex:3741: Warning: Application of '\'' on 'h' failed
```

Observe that using input encodings is a convenient alternative to accent commands — see Section B.17.4.

B.4 Sectioning

B.4.1 Sectioning Commands

Sectioning commands from `\part` down to `\subparagraph` are defined in base style files. They accept an optional argument and have starred versions.

The non-starred sectioning commands from `\part` down to `\subsubsection` show section numbers in sectional unit headings, provided their *level* is greater than or equal to the current value of the `secnumdepth` counter. Sectional unit levels and the default value of the `secnumdepth` counter are the same as in LaTeX. Furthermore, given a sectional unit *secname*, the counter *secname* exists and the appearance of sectional units numbers can be changed by redefining `\the`*secname*. For instance, the following redefinition turn the numbering of chapters into alphabetic (uppercase) style:

```
\renewcommand{\thechapter}{\Alph{chapter}}
```

When jumping to anchors, browsers put the targeted line on top of display. As a consequence, in the following code:

```
\section{A section}
\label{section:section}
 ...
See Section~\ref{section:section}
```

Clicking on the link produced by `\ref{section:section}` will result in *not* displaying the targeted section title. A fix is writing:

```
\section{\label{section:section}A section}
 ...
See Section~\ref{section:section}
```

Starting with version 2.04, HeVeA and HaCha will use the label name (`section:section` above) for the table of contents they generate. Notice that this behaviour applies to the `\label` command that occurs first in the sectioning command argument.

B.4.2 The Appendix

The `\appendix` command exists and should work as in LaTeX.

B.4.3 Table of Contents

HeVeA now generates a table of contents, using a procedure similar to the one of LaTeX(a `.htoc` file is involved). One inserts this table of contents in the main document by issuing the command `\tableofcontents`. Table of contents is controlled by the counter `tocdepth`. By default, the table of contents shows sectioning units down to the subsubsection level in *article* style and down to the subsection level in *book* (or *report*) style. To include more or less sectioning units in the table of contents, one should increase or decrease the `tocdepth` counter. It is important to notice that HeVeA produces such a table of contents, only when it has total control over cross-references. More precisely, HeVeA cannot produce the table of contents when it reads

LaTeX-produced `.aux` files. Instead, it should read its own `.haux` files. This will naturally occur if no `.aux` files are present, otherwise these `.aux` files should be deleted, or HeVeA should be instructed not to read them with the command-line option `-fix` (see Sections B.11.1 and C.1.1.4).

One can also add extra entries in the table of contents by using the command `\addcontentslines`, in a way similar to LaTeX homonymous command. However, hyperlinks need to be introduced explicitly, as in the following example, where an anchor is defined in the section title and referred to in the argument to `\addcontentsline`:

```
\subsection*{\aname{no:number}{Use \hacha{}}}
\addcontentsline{toc}{subsection}{\ahrefloc{no:number}{Use \hacha{}}}
```

(See Section 8.1.1 for details on commands related to hyperlinks.)

There is no list of figures nor list of tables.

Use HaCha

However, HeVeA has a more sophisticated way of producing a kind of map w.r.t. the sectioning of the document. A later run of HaCha on HeVeA output file splits it in smaller files organized in a tree whose nodes are tables of links. By contrast with LaTeX, starred sectioning commands generate entries in these tables of contents. Table of contents entries hold the optional argument to sectioning commands or their argument when there is no optional argument. Section 7 explains how to control HaCha.

B.5 Classes, Packages and Page Styles

B.5.1 Document Class

Both LaTeX 2_ε `\documentclass` and old LaTeX `\documentstyle` are accepted. Their argument *style* is interpreted by attempting to load a *style*`.hva` file. Presently, only the style files `article.hva`, `seminar.hva`, `book.hva` and `report.hva` exist, the latter two being equivalent.

If one of the recognized styles has already been loaded at the time when `\documentclass` or `\documentstyle` is executed, then no attempt to load a style file is made. This allows to override the document style file by giving one of the four recognized style files of HeVeA as a command line argument (see 2.2).

Conversely, if HeVeA attempt to load *style*`.hva` fails, then a fatal error is flagged, since it can be sure that the document cannot be processed.

B.5.2 Packages and Page Styles

HeVeA reacts to `\usepackage[`*options*`]{`*pkg*`}` in the following way:

1. The whole `\usepackage` command with its arguments gets echoed to the *image* file (see 6).
2. HeVeA attempt to load file *pkg*`.hva`, (see section C.1.1.1 on where HeVeA searches for files).

Note that HeVeA will not fail if it cannot load *pkg*`.hva` and that no warning is issued in that case.

The HeVeA distribution contains implementations of some packages, such as `verbatim`, `colors`, `graphics`, etc.

In some situations it may not hurt at all if HeVeA does not implement a package, for instance HeVeA does not provide an implementation for the `fullpage` package.

Users needing an implementation of a package that is widely used and available are encouraged to contact the author. Experienced users may find it fun to attempt to write package implementations by themselves.

B.5.3 The Title Page and Abstract

All title related commands exist, with the following peculiarities:

- The argument to the `\title` command appears in the HTML document header. As a consequence, titles should remain simple. Normal design (as regards HeVeA) is for `\title` to occur in the document preamble, so that the title is known at the time when the document header is emitted (while processing `\begin{document}`). However, there are two subtleties.

  If no `\title` command occurs in document preamble and that one `\title` command appears in the document, then the title is saved into the `.haux` file for a next run of HeVeA to put it in the HTML document header.

  If `\title` commands are present both in preamble and after `\begin{document}`, then the former takes precedence.

- When not present the date is left empty. The `\today` command generates will work properly only if `hevea` is invoked with the `-exec xxdate.exe` option. Otherwise `\today` generates nothing and a warning is issued.

The `abstract` environment is present is all base styles, including the *book* style. The `titlepage` environment does nothing.

B.6 Displayed Paragraphs

Displayed-paragraph environments translate to block-level elements.

In addition to the environments described in this section, HeVeA implements the `center`, `flushleft` and `flushright` environments. HeVeA also implements the corespondant TeX style declaration `\centering` `\raggedright` and `\raggedleft`, but these declarations may not work as expected, when they do not appear directly inside a displayed-paragraph environment or inside an array element.

B.6.1 Quotation and Verse

The `quote` and `quotation` environments are the same thing: they translate to `BLOCKQUOTE` elements. The `verse` environment is not supported.

B.6.2 List-Making environments

The `itemize`, `enumerate` and `description` environments translate to the `ul`, `ol`, and `DL` elements and this is the whole story.

As a consequence, no control is allowed on the appearances of these environments. More precisely optional arguments to `\item` do not function properly inside `itemize` and `enumerate`. Moreover, item labels inside `itemize` or numbering style inside `enumerate` are browser dependent.

However, customized lists can be produced by using the the `list` environment (see next section).

B.6.3 The `list` and `trivlist` environments

The `list` environment translates to the `DL` element. Arguments to `\begin{list}` are handled as follows:

`\begin{list}{`*default_ label*`}{`*decls*`}`

The first argument *default_ label* is the label generated by an `\item` command with no argument. The second argument, *decls* is a sequence of declarations. In practice, the following declarations are relevant:

`\usecounter{`*counter*`}` The counter *counter* is incremented by `\refstepcounter` by every `\item` command with no argument, before it does anything else.

`\renewcommand{\makelabel}[1]{...}` The command `\item` executes `\makelabel{`*label*`}`, where *label* is the item label, to print its label. Thus, users can change label formatting by redefining `\makelabel`. The default definition of `\makelabel` simply echoes *label*.

As an example, a list with an user-defined counter can be defined as follows:

```
\newcounter{coucou}
\begin{list}{\thecoucou}{%
\usecounter{coucou}%
\renewcommand{\makelabel}[1]{\textbf{#1}.}}
...
\end{list}
```

This yields:

1. First item.

2. Second item.

The `trivlist` environment is also supported. It is equivalent to the `description` environment.

B.6.4 Verbatim

The `verbatim` and `verbatim*` environments translate to the `PRE` element. Inside `verbatim*`, spaces are replaced by underscores ("␣").

Similarly, `\verb` and `\verb*` translate to the `CODE` text element.

The `alltt` environment is supported.

B.7 Mathematical Formulae

B.7.1 Math Mode Environment

The three ways to use math mode (`$...$`, `\(...\)` and `\begin{math}...\end{math}`) are supported. The three ways to use display math mode (`$$...$$`, `\[...\]` and `\begin{displaymath}... \end{displaymath}`) are also supported. Furthermore, `\ensuremath` behaves as expected.

The `equation`, `eqnarray`, `eqnarray*` environments are supported. Equation labelling and numbering is performed in the first two environments, using the `equation` counter. Additionally, numbering can be suppressed in one row of an `eqnarray`, using the `\nonumber` command.

Math mode is not as powerful in HeVeA as in LaTeX. The limitations of math mode can often be surpassed by using math display mode. As a matter of fact, math mode is for in-text formulas. From the HTML point of view, this means that math mode does not close the current flow of text and that formulas in math mode must be rendered using text-level elements only. By contrast, displayed formulas can be rendered using block-level elements. This means that HeVeA have much more possibilities in display context than inside normal flow of text. In particular, stacking text elements one above the over is possible only in display context.

B.7.2 Common Structures

HeVeA admits, subscript (`_`), superscripts (`^`) and fractions (`\frac{`*numer*`}{`*denom*`}`). The best effect is obtained in display mode, where HTML `table` element is extensively used. By contrast, when not in display mode, HeVeA uses only `SUB` and `SUP` text-level elements to render superscrits and subscript, and the result may not be very satisfying.

However, simple subscripts and superscripts, such as `x_i` or `x^2`, are always rendered using the `SUB` and `SUP` text-level elements and their appearance should be correct even in in-text formulas.

When occurring outside math mode, characters _ and ^ act as ordinary characters and get echoed to the output. However, a warning is issued.

An attempt is made to render all ellipsis constructs (`\ldots`, `\cdots`, `\vdots` and `\ddots`). The effect may be strange for the latter two.

B.7.3 Square Root

The n^{th} root command `\sqrt` is supported only for n=3,4, thanks to the existence of Unicode characters for the same. For the others, we shift to fractional exponents, in which case, the `\sqrt` command is defined as follows:

```
\newcommand{\sqrt}[3][2]{\left(#2\right)^{1/#1}}
```

B.7.4 Unicode and mathematical symbols

The support for unicode symbols offered by modern browsers allows to translate almost all math symbols correctly.

Log-like functions and variable sized-symbols are recognized and their subscripts and superscripts are put where they should in display mode. Subscript and superscript placement can be changed using the `\limits` and `\nolimits` commands. Big delimiters are also handled.

B.7.5 Putting one thing above/below/inside

The commands `\stackrel`, `\underline` and `\overline` are recognized. They produce sensible output in display mode. In text mode, these macros call the `\textstackrel`, `\textunderline` and `\textoverline` macros. These macros perform the following default actions

`\textstackrel` Performs ordinary superscripting.

`\textunderline` Underlines its argument, using the `U` text-level element.

`\textoverline` Overlines using style-sheets (used `<SPAN>` with a top border).

The command `\boxed` works well both in display and normal math mode. Input of the form `\boxed{\frac{\pi}{2}}` produces $\boxed{\frac{\pi}{2}}$ in normal math, and

$$\boxed{\frac{\pi}{2}}$$

in display-math mode. The commands `\bigl,\bigr` etc. are also rendered well. Some examples can be found in the test file `random-math.html` provided with the distribution.

B.7.6 Math accents

Math accents that have coresponding text accents (`\hat`, `\tilde`, etc.) are handled by default. They in fact act as the corresponding text-mode accents (Section B.3.4). As a consequence, they work properly only on ascii letters. This may be quite cumbersome, but at least some warnings draw user's attention on the problem. If accents are critical to your document and that HeVeA issues a lot of warnings, a solution is to redefine the math accent command. A suggested replacement is using limit superscripts. That way accents are positioned above symbols in display mode and after symbols in text mode.

```
\renewcommand{\hat}[1]{\mathop{#1}\limits^{\textasciicircum}\nolimits}
Displayed:
$$
```

```
\hat{\mu} = \hat{\Delta}.
$$
In text: $\hat{\mu} = \hat{\delta}$
```

The `\vec` command is rendered differently in display and non-display mode. In display mode, the arrow appears in normal position, while in non-display the arrow appears as an ordinary superscript.

`\vec{u}` in text mode: $\vec{u}$, `\vec{u}` in display mode: $\vec{u}$

Most "extensible accents" (`\widetilde`, `\widehat`, etc.) are not even defined. There are a few exceptions: line "accents":

$\underline{abc}$ `\underline` $\overline{abc}$ `\overline`

Brace "accents":

$\underbrace{1 \times 2 \times \cdots \times n}$ `\underbrace` $\overbrace{1 \times 2 \times \cdots \times n}$ `\overbrace`

And arrow "accents":

$\overleftarrow{1 \times 2 \times \cdots \times n}$ `\overleftarrow` $\overrightarrow{1 \times 2 \times \cdots \times n}$ `\overrightarrow`

B.7.7 Spacing

By contrast with LaTeX, space in the input matters in math mode. One or more spaces are translated to one space. Furthermore, spaces after commands (such as `\alpha`) are echoed except for invisible commands (such as `\tt`). This allows users to control space in their formulas, output being near to what can be expected.

Explicit spacing commands (`\,`, `\!`, `\:` and `\;`) are recognized, the first two commands do nothing, while the others two output one space.

B.7.8 Changing Style

Letters are italicized inside math mode and this cannot be changed. The appearance of other symbols can be changed using LaTeX 2_ε style changing commands (`\mathbf`, etc.). The commands `\boldmath` and `\unboldmath` are not recognized. Whether symbols belonging to the symbol font are affected by style changes or not is browser dependent.

The `\cal` declaration and the `\mathcal` command (that yield calligraphic letters in LaTeX) exist. They yield red letters by default.

Observe that this does not corresponds directly to how LaTeX manage style in math mode and that, in fact, style cannot really change in math mode.

Math style changing declarations `\displaystyle` and `\textstyle` do nothing when HeVeA is already in the requested mode, otherwise they issue a warning. This is so because HeVeA implements displayed maths as tables, which require to be both opened and closed and introduce line breaks in the output. As a consequence, warnings on `\displaystyle` are to be taken seriously.

The commands `\scriptstyle` and `\scriptscriptstyle` perform type size changes.

B.8 Definitions, Numbering

B.8.1 Defining Commands

HeVeA understands command definitions given in LaTeX style. Such definitions are made using `\newcommand`, `\renewcommand` and `\providecommand`. These three constructs accept the same arguments and have the same meaning as in LaTeX, in particular it is possible to define an user command with one optional argument. However, HeVeA is more tolerant: if command *name* already exists, then a subsequent `\newcommand{`*name*`}`... is

ignored. If macro *name* does not exists, then `\renewcommand{`*name*`}`... performs a definition of *name*. In both cases, LaTeX would crash, HeVeA just issues warnings.

The behaviour of `\newcommand` allows to shadow document definition, provided the new definitions are processed before the document definitions. This is easily done by grouping the shadowing definition in a specific style file given as an argument to HeVeA (see section 5.1). Conversely, changes of base macros (*i.e.* the ones that HeVeA defines before loading any user-specified file) must be performed using `\renewcommand`.

Scoping rules apply to macros, as they do in LaTeX. Environments and groups define a scope and command definition are local to the scope they occur.

It is worth noticing that HeVeA also partly implements TeX definitions (using `\def`) and bindings (using `\let`), see section B.16.1 for details.

B.8.2 Defining Environments

HeVeA accepts environment definitions and redefinitions by `\newenvironment` and `\renewenvironment`. The support is complete and should conform to [LaTeX, Sections C.8.2].

Environments define a scope both for commands and environment definitions.

B.8.3 Theorem-like Environments

New theorem-like environments can also be introduced and redefined, using `\newtheorem` and `\renewtheorem`.

Note that, by contrast with plain environments definitions, theorem-like environment definitions are global definitions.

B.8.4 Numbering

LaTeX counters are (fully ?) supported. In particular, defining a counter *cmd* with `\newcounter{`*cmd*`}` creates a macro `\the`*cmd* that outputs the counter value. Then the `\the`*cmd* command can be redefined. For instance, section numbering can be turned into alphabetic style by:

```
\renewcommand{\thesection}{\alph{section}}
```

Note that TeX style for counters is not supported at all and that using this style will clobber the output. However, HeVeA implements the *calc* package that makes using TeX style for counters useless in most situations (see section B.17.3).

B.8.5 The `ifthen` Package

The `ifthen` package is partially supported. The one unsupported construct is the `\lengthtest` test expression, which is undefined.

As a consequence, HeVeA accepts the following example from the LaTeX manual:

```
\newcounter{ca}\newcounter{cb}%
\newcommand{\printgcd}[2]{%
  \setcounter{ca}{#1}\setcounter{cb}{#2}%
  Gcd(#1,#2) =
  \whiledo{\not\(\value{ca}= \value{cb}\)}%
    {\ifthenelse{\value{ca}>\value{cb}}%
      {\addtocounter{ca}{-\value{cb}}}%
      {\addtocounter{cb}{-\value{ca}}}%
    gcd(\arabic{ca}, \arabic{cb}) = }%
  \arabic{ca}.}%
For example: \printgcd{54}{30}
```

For example: Gcd(54,30) = gcd(24, 30) = gcd(24, 6) = gcd(18, 6) = gcd(12, 6) = gcd(6, 6) = 6.

Additionally, a few boolean registers are defined by HeVeA. Some of them are of interest to users.

`hevea` Initial value is `true`. The `hevea.sty` style file also defines this register with initial value *false*.

`mmode` This register value reflects HeVeA operating mode, it is *true* in math-mode and *false* otherwise.

`display` This register value reflects HeVeA operating mode, it is *true* in display-mode and *false* otherwise.

`footer` Initial value is `true`. When set false, HeVeA does not insert its footer "*This document has been translated by HeVeA*".

Finally, note that HeVeA also recognised à la TEX conditional macros (see section B.16.1.4). Such macros are fully compatible with the boolean registers of the `ifthen` package, as it is the case in LaTeX.

B.9 Figures and Other Floating Bodies

Figures and tables are put where they appear in source, regardless of their placement arguments. They are outputted inside a `BLOCKQUOTE` element and they are separated from enclosing text by two horizontal rules.

Captions and cross referencing are handled. However captions are not moved at end of figures: instead, they appear where the `\caption` commands occur in source code. The `\suppressfloats` command does nothing and the figure related counters (such as `topnumber`) exist but are useless.

get marginal tes in the t margin, use eversemaginpar

Marginal notes are handled in an HeVeA specific way. By default, all notes go in the right margin. Issuing `\reversemarginpar` causes the notes to go in the left margin. Unsurprisingly, issuing `\normalmarginpar` reverts to default behaviour.

Marginal no go in the rig margin by d fault.

The `\marginpar` command has an optional argument.

`\marginpar[`*left_ text*`]{`*right_ text*`}`

If optional argument *left_ text* is present and that notes go in the left margin, then *left_ text* is the text of the note. Otherwise, *right_ text* is the text of the note. As a conclusion, marginal notes in HeVeA always go to a fixed side of the page, which side being controlled by the commands `\normalmarginpar` (right side) and `\reversemarginpar` (left side). This departs form LaTeX that selects a default side depending on the parity of the page counter.

Marginal notes are styled by the means of two environment style classes (see Section 9.3) : `marginpar` and `marginparside`. The latter `marginparside` takes care of margins and placement as a float, its value is `marginparright` for notes in the right margin and `marginparleft` for notes in the left margin. Users are not expected to alter those. The `marginpar` environment style class governs the general aspect of all marginal notes. Users can control the aspect of all marginal notes by defining a new style class and assigning the `marginpar` environment style class. For instance, to get all marginal notes in red font, and taking 10% of the page width (in place of the default 20%), one can issue the following commands in the document preamble.

```
\newstyle{.mynote}{width:10\%; color:red;}
\setenvclass{marginpar}{mynote}
```

B.10 Lining It Up in Columns

B.10.1 The `tabbing` Environment

Limited support is offered. The `tabbing` environment translate to a flexible `tabular`-like environment. Inside this environment, the command `\kill` ends a row, while commands `\=` and `\>` start a new column. All other tabbing commands do not even exist.

B.10.2 The `array` and `tabular` environments

These environments are supported, using HTML `table` element, rendering is satisfactory in most (not too complicated) cases. By contrast with LaTeX, some of the array items always are typeset in display mode. Whether an array item is typeset in display mode or not depends upon its column specification, the `l`, `c` and `r` specifications open display mode while the remaining `p` and `@` do not. The `l`, `c`,`r` and `@` specifications disable word wrap, while the `p` specification enables it.

Entries in a column whose specification is `l` (resp. `c` or `r`) get left-aligned (resp. centered or right-aligned) in the horizontal direction. They will get top-aligned in the vertical direction if there are other column specifications in the same array that specify vertical alignment constraints (such as `p{`*wd*`}`, see below). Otherwise, vertical alignment is unspecified.

Entries in a column whose specification is `p{`*wd*`}` get left-aligned in the horizontal direction and top-aligned in the vertical direction and a paragraph break reduces to one line break inside them. This is the only occasion where HeVeA makes a distinction between LR-mode and paragraph mode. Also observe that the length argument *wd* to the `p` specification is ignored.

Some LaTeX array features are not supported at all:

- Optional arguments to `\begin{array}` and `\begin{tabular}` are ignored.
- The command `\vline` does not exists.

Some others are partly rendered:

- Spacing between columns is different.
- `@` formatting specifications in `\multicolumn` argument are ignored.
- If a `|` appears somewhere in the column formatting specification, then the array is shown with borders.
- The command `\hline` does nothing if the array has borders (see above). Otherwise, an horizontal rule is outputted.
- The command `\cline` ignores its argument and is equivalent to `\hline`.
- Similarly the command `\extracolsep` issues a warning and ignores its argument.

Additionally, the `tabular*` environment is recognised and gets rendered as an HTML table with an advisory width attribute.

By default, HeVeA implements the `array` package (see [LaTeX-bis, Section 5.3] and section B.17.2 in this document), which significantly extends the `array` and `tabular` environments.

B.11 Moving Information Around

B.11.1 Files

In some situations, HeVeA uses some of the ancillary files generated by LaTeX. More precisely, while processing file *doc*`.tex`, the following files may be read:

`.aux` The file *doc*`.aux` contains cross-referencing information, such as figure or section numbers. If this file is present, HeVeA reads it and put such numbers (or labels) inside the links generated by the `\ref` command. If the `.aux` file is not present, or if the `hevea` command is given the `-fix` option, HeVeA will instead use `.haux` files.

`.haux` Such files are HeVeA equivalents of `.aux` files. Indeed, they are `.aux` files tailored to HeVeA needs. Two runs of HeVeA might be needed to get cross references right.

`.htoc` This file contains a formatted table of contents. It is produced while reading the `.haux` file. As consequence a table of contents is available only when the `.haux` file is read.

`.hbbl` The *doc*`.hbbl` file is generated by `bibhva` from *doc*`.haux`. When present, it is read by the `\bibliography` command.

`.bbl` The *doc*`.bbl` file is generated by BibTeX from *doc*`.aux`. When present, and if no *doc*`.hbbl` exists, *doc*`.bbl` is read by the `\bibliography` command.

`.hidx` **and** `.hind` HeVeA computes its own indexes, using `.hidx` files for storing index references and, using `.hind` files for storing formatted indexes. Index formatting significantly departs from the one of LaTeX. Again, several runs of HeVeA might be needed to get indexes right.

HeVeA does not fail when it cannot find an auxiliary file. When another run of HeVeA is needed, a warning is issued, and it is user's responsibility to rerun HeVeA. However, the convenient `-fix` command-line option instructs HeVeA to rerun itself, until it believes it has reached stable state.

B.11.2 Cross-References

The LaTeX commands `\label` and `\ref` are changed by HeVeA into HTML anchors and local links, using the "`a`" element. Additionally, numerical references to sectional units, figures, tables, etc. are shown, as they would appear in the `.dvi` file. Numerical references to pages (such as generated by `\pageref`) are not shown; only an link is generated.

The anchor used is the *label* argument to `\label{`*label*`}`. More precisely, `\label{`*label*`}` translates to `<a id="`*label*`"></a>`; while `\ref{`*label*`}` translates to `<a name="#`*label*`">`*nnn*`</a>`, where *nnn* is the appropriate numerical reference to a section. As a consequence spaces are better avoided in *label*.

Starting with HeVeA version 2.04, the HTML anchors used by `\label` and `\ref` cannot differ from the arguments to these commands anymore. Moreover, when `\label{`*label*`}` occurs inside the argument of a sectioning command (*i.e.* in section title, as recommended by section B.4.1), then HeVeA and HaCha will use *label* as the "`id`" attribute of the corresponding section. For instance, the LaTeX source of this very section is:

```
\subsection{Cross-References\label{cross-reference}}
```

It translates to HTML similar to

```
<h3 class="subsection" id="cross-reference">B.11.2  Cross-References</h3>
```

Notice that no `<a id="cross-reference"></a>` appears above. Instead `id="cross-reference"` appears in the enclosing `h3` header element.

While processing a document *doc*`.tex`, cross-referencing information can be computed in two different, mutually exclusive, ways, depending on whether LaTeX has been previously run or not:

- If there exists a file *doc*`.aux` and that `hevea` has not been given the command-line option `-fix`, then cross-referencing information is extracted from that file. Of course, the *doc*`.aux` file has to be up-to-date, that is, it should be generated by running LaTeX as many times as necessary. (For HeVeA needs, one run is probably sufficient).

- If no *doc*`.aux` file exists or if `hevea` has been given the `-fix` command-line option, then HeVeA expect to find cross-referencing information in the file *doc*`.haux`.

The second option is recommended.

When using its own *doc*`.haux` file, HeVeA will output a new *doc*`.haux` file at the end of its processing. This new *doc*`.haux` file contains actualised cross referencing information. Hence, in that case, HeVeA may need to run twice to get cross-references right. Note that, just like LaTeX, HeVeA issues a warning then the

cross-referencing information it generates differs from what it has read at start-up, and that it does not fail if *doc*`.haux` does not exist.

Observe that if a non-correct *doc*`.aux` file is present, then cross-references will apparently be wrong. However the links are correct.

B.11.3 Bibliography and Citations

The `\cite` macro is supported. Its optional argument is correctly handled. Citation labels are extracted from the `.aux` file if present, from the `.haux` file otherwise. Note that these labels are put there by LaTeX in the first case, and by HeVeA in the second case, when they process the `\bibitem` command.

Using BibTeX

All BibTeX related commands exist and echo the appropriate information into the `.haux` file.

In particular, the `\bibliography` command exists and attempts to load the formatted bibliography, *i.e.* to load the `.hbbl` file. The `.hbbl` file is produced from the `.haux` file by the companion program `bibhva` (see C.1.4). To include the bibliographic references extracted from `.bib` databases, it should normally suffice to do:

```
# hevea doc.tex
# bibhva doc
# hevea doc.tex
```

In case no `.hbbl` file exists, the `\bibliography` command attempts to load the `.bbl` file normally used while combining LaTeX and BibTeX. Thus, another way to extract bibliographic references from `.bib` databases is:

```
# latex doc.tex
# bibtex doc
# hevea doc.tex
```

In case both files exist, notice that loading the `.hbbl` file has priority over loading the `.bbl` file.

B.11.4 Splitting the Input

The `\input` and `\include` commands exist and they perform exactly the same operation of searching (and then processing) a file, whose name is given as an argument. See section C.1.1.1 on how HeVeA searches files. However, in the case of the `\include` command, the file is searched only when previously given as an argument to the `\includeonly` command.

Note the following features:

- TeX syntax for `\input` is not supported. That is, one should write `\input{`*filename*`}`.
- If *filename* is excluded with the `-e` command-line option (see section C.1.1.4), then HeVeA does not attempt to load *filename*. Instead, it echoes `\input{`*filename*`}` and `\include{`*filename*`}` commands into the *image* file. This sounds complicated, but this is what you want!
- HeVeA does not fail when it cannot find a file, it just issues a warning.

The `\listfiles` command is a null command.

B.11.5 Index and Glossary

Glossaries are not handled (who uses them ?).

While processing a document *doc*.`tex`, index entries go into the file *doc*.`hidx`, while the formatted index gets written into the file *doc*.`hind`. As with LaTeX, two runs of HeVeA are normally needed to format the index. However, if all index producing commands (normally `\index`) occur before the index formatting command (normally `\printindex`), then only one run is needed.

As in LaTeX, index processing is not enabled by default and some package has to be loaded explicitly in the document preamble. To that aim, HeVeA provides the standard package `makeidx`, and two extended packages that allow the production of several indexes (see section B.17.7).

Formatting of indexes in HeVeA departs from LaTeX behaviour. More precisely the `theindex` environment does not exist. Instead, indexes are formatted using special `indexenv` environments. Those details do not normally concern users. However, the number of columns in the presentation of the index can be controlled by setting the value of the `indexcols` counter (default value is two).

B.11.6 Terminal Input and Output

The `\typeout` command echos its argument on the terminal, macro parameter `#`i are replaced by their values. The `\typein` command is not supported.

B.12 Line and Page Breaking

B.12.1 Line Breaking

The advisory line breaking command `\linebreak` will produce a line break if it has no argument or if its optional argument is 4. The `\nolinebreak` command is a null command.

The `\\` and `\\*` commands output a `<BR>` tag, except inside arrays where the close the current row. Their optional argument is ignored. The `\newline` command outputs a `<BR>` tag.

All other line breaking commands, declarations or environments are silently ignored.

B.12.2 Page Breaking

They are no pages in the physical sense in HTML. Thus, all these commands are ignored.

B.13 Lengths, Spaces and Boxes

B.13.1 Length

All length commands are ignored, things go smoothly when LaTeX syntax is used (using the `\newlength`, `\setlength`, etc. commands, which are null macros). Of course, if lengths are really important to the document, rendering will be poor.

Note that TeX length syntax is not at all recognised. As a consequence, writing things like `\textwidth=10cm` will clobber the output. Users can correct such misbehaviour by adopting LaTeX syntax, here they should write `\setlength{\textwidth}{10cm}`.

B.13.2 Space

The `\hspace`, `\vspace` and `\addvspace` spacing commands and their starred versions recognise positive explicit length arguments. Such arguments get converted to a number of non-breaking spaces or line breaks. Basically, the value of `1em` or `1ex` is one space or one line-break. For other length units, a simple conversion based upon a 10pt font is used.

HeVeA cannot interpret more complicated length arguments or perform negative spacing. In these situations, a warning is issued and no output is done.

Spacing commands without arguments are recognised. The `\enspace`, `\quad` and `\qquad` commands output one, two and four non-breaking spaces, while the `\smallskip`, `\medskip` and `\bigskip` output one, one, and two line breaks.

Stretchable lengths do not exist, thus the `\hfill` and `\vfill` macros are undefined.

B.13.3 Boxes

Box contents is typeset in text mode (*i.e.* non-math and non-display mode). Both LaTeX boxing commands `\mbox` and `\makebox` commands exist. However `\makebox` generates a specific warning, since HeVeA ignore the length and positioning instructions given as optional argument.

Similarly, the boxing with frame `\fbox` and `\framebox` commands are recognised and `\framebox` issues a warning. When in display mode, `\fbox` frames its argument by enclosing it in a table with borders. Otherwise, `\fbox` calls the `\textfbox` command, which issues a warning and typesets its argument inside a `\mbox` (and thus no frame is drawn). Users can alter the behaviour of `\fbox` in non-display mode by redefining `\textfbox`.

Boxes can be saved for latter usage by storing them in *bins*. New bins are defined by `\newsavebox{`*cmd*`}`.

Then some text can be saved into *cmd* by `\sbox{`*cmd*`}{`*text*`}` or `\begin{lrbox}{`*cmd*`}` *text* `\end{lrbox}`. The text is translated to HTML, as if it was inside a `\mbox` and the resulting output is stored. It is retrieved (and outputted) by the command `\usebox{`*cmd*`}`. The `\savebox` command reduces to `\sbox`, ignoring its optional arguments.

The `\rule` commands translate to a HTML horizontal rule (`<HR>`) regardless of its arguments.

All other box-related commands do not exist.

B.14 Pictures and Colours

B.14.1 The `picture` environment and the `graphics` Package

It is possible to have pictures and graphics processed by `imagen` (see section 6.1). In the case of the `picture` environment it remains users responsibility to explicitly choose source chunks that will get rendered as images. In the case of the commands from the `graphics` package, this choice is made by HeVeA.

For instance consider the following picture:

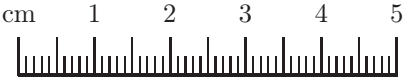
```
\newcounter{cms}
\setlength{\unitlength}{1mm}
\begin{picture}(50,10)
\put(0,7){\makebox(0,0)[b]{cm}}
\multiput(10,7)(10,0){5}{\addtocounter{cms}{1}\makebox(0,0)[b]{\arabic{cms}}}
\multiput(1,0)(1,0){49}{\line(0,1){2.5}}
\multiput(5,0)(10,0){5}{\line(0,1){5}}
\thicklines
\put(0,0){\line(1,0){50}}
\multiput(0,0)(10,0){6}{\line(0,1){5}}
\end{picture}
```

Users should enclose *all* picture elements in a `toimage` environment (or inside `%BEGIN IMAGE... %END IMAGE` comments) and insert an `\imageflush` command, where they want the image to appear in HTML output:

```
%BEGIN IMAGE
\newcounter{cms}
\setlength{\unitlength}{1mm}
```

```
\begin{picture}(50,10)
  ...
\end{picture}
%END IMAGE
%HEVEA\imageflush
```

This will result in normal processing by LaTeX and image inclusion by HeVeA:



All commands from the graphics package are implemented using the automatic image inclusion feature. More precisely, the outermost invocations of the `\includegraphics`, `\scalebox`, etc. commands are sent to the image *image* file and there will be one image per outermost invocation of these commands.

For instance, consider a document `doc.tex` that loads the `graphics` package and that includes some (scaled) images by:

```
\begin{center}
\scalebox{.5}{\includegraphics{round.ps}}
\scalebox{.75}{\includegraphics{round.ps}}
\includegraphics{round.ps}
\end{center}
```

Then, issuing the following two commands:

```
# hevea doc.tex
# imagen doc
```

yields HTML that basically consists in three image links, the images being generated by `imagen`. Since the advent of `pdflatex`, using `\includegraphics` to insert bitmap images (*e.g.* `.gif` or `.jpg`) became frequent. In that case, users are advised *not* to use HeVeA default implementation of the `graphics` package. Instead, they may use a simple variation of the technique described in Section 8.2.

B.14.2 The `color` Package

HeVeA partly implements the `color` package. Implemented commands are `\definecolor`, `\color`, `\colorbox`, `\textcolor`, `\colorbox` and `\fcolorbox`. Other commands do not exist. At startup, colours `black`, `white`, `red`, `green`, `blue`, `cyan`, `yellow` and `magenta` are pre-defined.

Colours are defined by `\definecolor{`*name*`}{`*model*`}{`*spec*`}`, where *name* is the color name, *model* is the color model used, and *spec* is the color specification according to the given model. Defined colours are used by the declaration `\color{`*name*`}` and by the command `\textcolor{`*name*`}{`*text*`}`, which change text color. Please note that, the `\color` declaration accepts color specifications directly when invoked as `\color[`*model*`]{`*spec*`}`. The `\textcolor` command has a similar feature.

As regards color models, HeVeA implements the `rgb`, `cmyk`, `hsv` and `hls` color models. In those models, color specifications are floating point numbers less than one. For instance, here is the definition for the `red` color:

```
\definecolor{red}{rgb}{1, 0, 0}
```

The `named` color model is also supported, in this model color specification are just names... Named colours are the ones of `dvips`.

GreenYellow, Yellow, Goldenrod, Dandelion, Apricot, Peach, Melon, YellowOrange, Orange, BurntOrange, Bittersweet, RedOrange, Mahogany, Maroon, BrickRed, Red, OrangeRed, RubineRed, WildStrawberry, Salmon, CarnationPink, Magenta, VioletRed, Rhodamine, Mulberry, RedViolet, Fuchsia, Lavender, Thistle, Orchid, DarkOrchid, Purple, Plum, Violet, RoyalPurple, BlueViolet, Periwinkle, CadetBlue,

CornflowerBlue, MidnightBlue, NavyBlue, RoyalBlue, Blue, Cerulean, Cyan, ProcessBlue, SkyBlue, Turquoise, TealBlue, Aquamarine, BlueGreen, Emerald, JungleGreen, SeaGreen, Green, ForestGreen, PineGreen, LimeGreen, YellowGreen, SpringGreen, OliveGreen, RawSienna, Sepia, Brown, Tan, Gray, Black, White.

There are at least three ways to use colours from the `named` model.

1. Define a color name for them.
2. Specify the named color model as an optional argument to `\color` and `\textcolor`.
3. Use the names directly (HeVeA implements the `color` package with the `usenames` option given).

That is:

1. `\definecolor{rouge-brique}{named}{BrickRed}\textcolor{rouge-brique}{Text as a brick}`.
2. `\textcolor[named]{BrickRed}{Text as another brick}`.
3. `\textcolor{BrickRed}{Text as another brick}`.

HeVeA also implements the `\colorbox` and `\fcolorbox` commands.

```
\colorbox{red}{Red background},
\fcolorbox{magenta}{red}{Red background, magenta border}.
```

Red background, Red background, magenta border.

Those two commands accept an optional first argument that specifies the color model, as `\textcolor` does:

```
\fcolorbox[named]{RedOrange}{Apricot}{Apricot background, RedOrange border}.
```

Apricot background, RedOrange border.

Colours should be used carefully. Too many colours hinders clarity and some of the colours may not be readable on the document background color.

B.14.2.1 The `bgcolor` environment

With respect to the LaTeX `color` package, HeVeA features an additional `bgcolor` environment, for changing the background color of some subparts of the document. The `bgcolor` environment is a displayed environment and it normally starts a new line. Simple usage is `\begin{bgcolor}{`*color*`}`... `\end{bgcolor}`, where *color* is a color defined with `\definecolor`. Hence the following source yield a paragraph with a red background:

```
\begin{bgcolor}{red}
\color{yellow}Yellow letters on a red backgroud
\end{bgcolor}
```

The `bgcolor` environment is implemented by one-cell `table` element, it takes an optional argument that is used as an attribute for the inner `td` element (default value is `style="padding:1em"`). Advanced users may change the default, for instance as:

```
\begin{bgcolor}[style="padding:0"]{yellow}
\color{red}Red letters on a yellow backgroud
\end{bgcolor}
```

The resulting output will be red letters on a yellow background and no padding:

B.14.2.2 From High-Level Colours to Low-Level Colours

High-level colours are color names defined with `\definecolor`. Low-level colours are HTML-style colours. That is, they are either one of the sixteen conventional colours black, silver etc., or a RGB hexadecimal color specification of the form `"#XXXXXX"`.

One changes the high-level *high-color* into a low-level color by `\@getcolor{`*high-color*`}`. Low-level colours are appropriate inside HTML attributes and as arguments to the `\@fontcolor` internal macro. An example of `\@getcolor` usage can be found at the end of section 8.5.

There is also `\@getstylecolor` command that acts like`\@getcolor`, except that it does not output the double quotes around RGB hexadecimal color specifications. Such low-level colours are appropriate for style definitions in cascading style sheets [CSS-2]. See Section 9.3 for an example.

B.15 Font Selection

B.15.1 Changing the Type Style

All LaTeX 2_ε declarations and environments for changing type style are recognised. Aspect is rather like LaTeX 2_ε output, but there is no guarantee.

As HTML does not provide the same variety of type styles as LaTeX does. However CSS provide a wide variety of font properties. HeVeA uses generic properties, proper rendering will then depend upon user agent. For instance, it belongs to the user agent to make a difference between *italics* (rendered by the font style "italic") and *slanted* (rendered by the font style "oblique").

Here is how HeVeA implements text-style declarations by default:

`\itshape`	font-style:italic
`\slshape`	font-style:oblique
`\scshape`	font-variant:small-caps
`\upshape`	no style

`\ttfamily`	font-family:monospace
`\sffamily`	font-family:sans-serif
`\rmfamily`	no style

`\bfseries`	font-weight:bold
`\mdseries`	no style

Text-style commands also exists, they are defined as `\mbox{\`*decl*`...}`. For instance, `\texttt` is defined as a command with one argument whose body is `\mbox{\ttfamily#1}`. Finally, the `\emph` command for emphasised text also exists, it yields text-level `em` elements.

As in LaTeX, type styles consists in three components: *shape*, *series* and *family*. HeVeA implements the three components by making one declaration to cancel the effect of other declarations of the same kind.

Old style declarations are also recognised, they translate to text-level elements. However, no elements are cancelled when using old style declaration. Thus, the source "`{\sl\sc slanted and small caps}`" yields "slanted" small caps. Users need probably not worry about this. However this has an important practical consequence: to change the default rendering of type styles, one should redefine old style declaration in order to benefit from the cancellation mechanism. See section 10.2 for a more thorough description.

B.15.2 Changing the Type Size

All declarations, from `\tiny` to `\Huge` are recognised. Output is not satisfactory inside headers elements generated by sectioning commands.

B.15.3 Special Symbols

The `\symbol{`*num*`}` outputs character number *num* (decimal) from the Unicode character set. This departs from LaTeX, which output symbol number *num* in the current font.

B.16 Extra Features

This section describes HeVeA functionalities that extends on plain LaTeX, as defined in [LaTeX]. Most of the features described here are performed by default.

B.16.1 TeX macros

Normally, HeVeA does not recognise constructs that are specific to TeX. However, some of the internal commands of HeVeA are homonymous to TeX macros, in order to enhance compatibility. Note that full compatibility with TeX is not guaranteed.

B.16.1.1 À la TeX macros definitions

The `\def` construct for defining commands is supported. It is important to notice that HeVeA semantics for `\def` follows TeX semantics. That is, defining a command that already exists with `\def` succeeds.

Delimiting characters in command definition are somehow supported. Consider the following example from the TeX Book:

```
\def\Look{\textsc{Look}}
\def\x{\textsc{x}}
\def\cs AB#1#2C$#3\$ {#3{ab#1}#1 c\x #2}
\cs AB {\Look}{}C${And \$}{look}\$ 5.
```

It yields: And $lookabLookLook cx5.

Please note that delimiting characters are supported as far as I could, problems are likely with delimiting characters which include spaces or command names, in particular the command name `\{`. One can include `\{` in a command argument by using the grouping characters `{... }`:

```
\def\frenchquote(#1){\guillemotleft~\emph{#1}~\guillemotright{} (in French)}
he said \frenchquote(Alors cette accolade ouvrante {``\{''}~?).
```

Yields: he said « *Alors cette accolade ouvrante "{" ?* » (in French).

Another issue regards comments: "%" in arguments may give undefined behaviours, while comments are better avoided while defining macros. As an example, the following code will *not* be handled properly by HeVeA:

```
\def\x%
   #1{y}
```

Such TeX source should be rewritten as `\def\x#1{y}`.

Another source of incompatibility with TeX is that substitution of macros parameters is not performed at the same moment by HeVeA and TeX. However, things should go smoothly at the first level of macro expansion, that is when the delimiters appear in source code at the same level as the macro that is to parse them. For instance, the following source will give different results in LaTeX and in HeVeA:

```
\def\cs#1A{``#1''}
\def\othercs#1{\cs#1A}
\othercs{coucouA}
```

LaTeX output is "coucou"A, while HeVeA output is "coucouA". For instance, here is LaTeX output: "coucou"A. Please note that in most situations this discrepancy will make HeVeA crash.

B.16.1.2 The `\let` construct

HeVeA also processes a limited version of `\let`:

`\let` *macro-name1* = *macro-name2*

The effect is to bind *macro-name1* to whatever *macro-name2* is bound to at the time `\let` is processed. This construct may prove very useful in situations where one wishes to slightly modify basic commands. See sections 10.3 and B.2 for examples of using `\let` in such a situation.

B.16.1.3 The `\global` construct

It is possible to escape scope and to make global definitions and bindings by using the TeX construct `\global`. The `\global` construct is significant before `\def` and `\let` constructs.

Also note that `\gdef` is equivalent to `\global\def`.

B.16.1.4 TeX Conditional Macros

The `\newif\if`*name*, where *name* is made of letters only, creates three macros: `\if`*name*, `\`*name*`true` and `\`*name*`false`. The latter two set the *name* condition to *true* and *false*, respectively. The `\if`*name* command tests the condition *name*:

`\if`*name*
text$_1$
`\else`
text$_2$
`\fi`

Text *text*$_1$ is processed when *name* is *true*, otherwise *text*$_2$ is processed. If *text*$_2$ is empty, then the `\else` keyword can be omitted.

Note that HeVeA also implements LaTeX `ifthen` package and that TeX simple conditional macros are fully compatible with LaTeX boolean registers. More precisely, we have the following correspondences:

TeX	LaTeX
`\newif\if`*name*	`\newboolean{`*name*`}`
`\`*name*`true`	`\setboolean{`*name*`}{true}`
`\`*name*`false`	`\setboolean{`*name*`}{false}`
`\if`*name* *text*$_1$`\else` *text*$_2$`\fi`	`\ifthenelse{\boolean{`*name*`}}{`*text*$_1$`}{`*text*$_2$`}`

B.16.1.5 Other TeX Macros

HeVeA implements the macros `\unskip` and `\endinput`. It also supports the `\csname`... `\endcsname` construct.

B.16.2 Command Definition inside Command Definition

If one strictly follows the LaTeX manual, only commands with no arguments can be defined inside other commands. Parameters (*i.e.* `#`*n*) occurring inside command bodies refer to the outer definition, even when they appear in nested command definitions. That is, the following source:

```
\newcommand{\outercom}[1]{\newcommand{\insidecom}{#1}\insidecom}
\outercom{outer}
```

yields this output:

outer

Nevertheless, nested commands with arguments are allowed. Standard parameters `#`n still refer to the outer definition, while nested parameters `##`n refer to the inner definition. That is, the source:

```
\newcommand{\outercom}[1]{\newcommand{\insidecom}[1]{##1}\insidecom{inner}}
\outercom{outer}
```

yields this output:

inner

B.16.3 Date and time

Date and time support is not enabled by default, for portability and simplicity reasons.

However, HeVeA source distribution includes a simple (`sh`) shell script `xxdate.exe` that activates date and time support. The `hevea` command, should be invoked as:

```
# hevea -exec xxdate.exe ...
```

This will execute the script `xxdate.exe`, whose output is then read by HeVeA. As a consequence, standard LaTeX counters `year`, `month`, `day` and `time` are defined and LaTeX command `\today` works properly. Additionally the following counters and commands are defined:

Counter `weekday`	day of week, 0...6
Counter `Hour`	hour, 00...11
Counter `hour`	hour, 00...23
Counter `minute`	minute, 00...59
Counter `second`	second, 00...61 (According to `date` man page!)
Command `\ampm`	AM or PM
Command `\timezone`	Time zone
Command `\heveadate`	Output of the `date` Unix command

Note that I chose to add an extra option (and not an extra `\@exec` primitive) for security reasons. You certainly do not want to enable HeVeA to execute silently an arbitrary program without being conscious of that fact. Moreover, the `hevea` program does not execute `xxdate.exe` by default since it is difficult to write such a script in a portable manner.

Windows users should enjoy the same features with the version of `xxdate.exe` included in the Win32 distribution.

B.16.4 Fancy sectioning commands

Loading the `fancysection.hva` file will radically change the style of sectional units headers: they appear over a green background, the background color saturation decreases as the sectioning commands themselves do. Additionally, the document background color is white.

Note : Fancy section has been re-implemented using style-sheets. While it respects the old behaviour, users are encouraged to try out style-sheets for more flexibility. See Section 9 for details.

The `fancysection.hva` file is intended to be loaded after the document base style. Hence the easiest way to load the `fancysection.hva` file is by issuing `\usepackage{fancysection}` in the document preamble. To allow processing by LaTeX, one may for instance create an empty `fancysection.sty` file.

As an alternative, to use fancy section style in `doc.tex` whose base style is *article* you should issue the command:

```
# hevea article.hva fancysection.hva doc.tex
```

You can also make a `doc.hva` file that contains the two lines:

```
\input{article.hva}
\input{fancysection.hva}
```

And then launch `hevea` as:

```
# hevea doc.hva doc.tex
```

Sectioning command background colours can be changed by redefining the corresponding colours (`part`, `chapter`, `section`,...). For instance, you get various mixes of red and orange by:

```
\input{article.hva}
\input{fancysection.hva}
\definecolor{part}{named}{BrickRed}
\definecolor{section}{named}{RedOrange}
\definecolor{subsection}{named}{BurntOrange}
```

(See section B.14.2 for details on the `named` color model that is used above.)

Another choice is issuing the command `\colorsections{`*hue*`}`, where *hue* is a hue value to be interpreted in the HSV model. For instance,

```
\input{article.hva}
\input{fancysection.hva}
\colorsections{20}
```

will yield sectional headers on a red-orange background.

HeVeA distribution features another style for fancy sectioning commands: the `undersection` package provides underlined sectional headers.

B.16.5 Targeting Windows

At the time of this release, Windows support for symbols through Unicode is not as complete as the one of Linux, which I am using for testing HeVeA.

One of the most salient shortcomings is the inability to display sub-elements for big brackets, braces and parenthesis, which HeVeA normally outputs when it processes `\left[`, `\right\}` etc.

We (hopefully) expect Windows fonts to display more of Unicode easily in a foreseeable future. As a temporary fix, we provide a style file `winfonts.hva`. Authors concerned by producing pages that do not look too ugly when viewed through Windows browsers are thus advised to load the file `winfonts.hva`. For instance they can invoke HeVeA as:

```
# hevea winfonts.hva ...
```

At the moment, loading `winfonts.hva` only changes the rendering of LaTeX big delimiters, avoiding the troublesome Unicode entities.

More generally, it remains authors responsibility to be careful not to issue too refined Unicode entities. To that aim, authors that target a wide audience should first limit themselves to the most common symbols (*e.g.* use `\leq` [$\leq$] in place of `\preceq` [$\preceq$]) and, above all, they should control the rendering of their documents using several browsers.

B.16.6 MathJax support

MathJax support is enabled by loading the mathjax package. Two operating mode modes are provided: explicit and automatic. Notice that HeVeA distribution includes a innocuous `mathjax.sty` for LaTeX compatibility — see also Sec. C.4.2.

B.16.6.1 Explicit mode

Explicit mode is enabled when `\usepackage{mathjax}` appears in the document preamble, or when HeVeA is invoked as "`hevea mathjax.hva`...".

Basic consists in one environment `displayjax` and one command `\textjax`. The environment is appropriate for displayed maths. As an example, the following source

```
A displayed formula:
\begin{displayjax}
\frac{\pi}{4} = \left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} +
\frac{1}{9} + \cdots + \frac{(-1)^n}{2n+1} + \cdots \right]
\end{displayjax}
```

is displayed as follows: A displayed formula:

$$\frac{\pi}{4} = \left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} + \cdots + \frac{(-1)^n}{2n+1} + \cdots \right]$$

The `\textjax` command is appropriate for inline mathematical contents. For instance, the following source

```
``A nice inline formula:
\textjax{\frac{\pi}{4} = \left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} +
\frac{1}{9} + \cdots + \frac{(-1)^n}{2n+1} + \cdots \right]}.''
```

is typeset as: "A nice inline formula: $\frac{\pi}{4} = \left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} + \cdots + \frac{(-1)^n}{2n+1} + \cdots \right]$."

Advanced support consists in the `mathjax` environment. Source code enclosed in `\begin{mathjax}\ldots\end{mathjax}` will be reproduced into output for the MathJax script to handle it. However, HeVeA does not start any other action. Thanks to this feature, users can have any (recognised by MathJax) displayed math environment processed by MathJax. For instance, the following source

```
\begin{mathjax}
\begin{eqnarray*}
z^2  & = & x^2 + y^2\\
\end{eqnarray*}
\end{mathjax}
```

will be displayed as:

$$z^2 \quad = \quad x^2 + y^2$$

Finally, notice that a document that uses the explicit MathJax constructs can be processed by LaTeX, provided it loads the `mathjax.sty` file present in HeVeA distribution. This can be done simply by having the line `\usepackage{mathjax}` in teh document preamble. Then, HeVeA and LaTeX will react appropriately (see sections 2.3.2 and B.5.2).

B.16.6.2 Automatic mode

Automatic mode is enabled when `\usepackage[auto]{mathjax}` appears in the document preamble, or when HeVeA is invoked as "`hevea mathjaxauto.hva`...".

In automatic mode, HeVeA will pass all mathematical text to MathJax. This mode seems by far the most practical, but beware:

1. There is no communication back from MathJax to HeVeA. As result, equation numbers, as generated for instance by the `equation` environment, will not find their way to the final display.
2. Some constructs, such as `\mbox`, are not handled by MathJax.

Table 1: Column specifications from the `array` package

`m{`*width*`}`	Equivalent to the `p` column specification (the *width* argument is ignored, entries are typeset in paragraph mode with paragraph breaks being reduced to a single line break), except that the entries are centered vertically.
`b{`*width*`}`	Equivalent to the `p` column specification, except that the entries are bottom-aligned vertically.
`>{`*decl*`}`	Can be used before `l`, `c`, `r`, `p{...}`, `m{...}` or `b{...}`. It inserts *decl* in front of the entries in the corresponding column.
`<{`*decl*`}`	Can be used after `l`, `c`, `r`, `p{...}`, `m{...}` or `b{...}`. It inserts *decl* after entries in the corresponding column.
`!{`*decl*`}`	Equivalent to `@{`*decl*`}`

B.16.6.3 Customising the MathJax script

By default HeVeA insert a reference to the "default" MathJax script with "default" configuration parameters. Advanced users can change this setting by redefining the `\jax@meta` command, which must contain the appropriate `<script>` element. See the file `html/mathjax.hva` for details.

B.17 Implemented Packages

HeVeA distribution includes `.hva` packages that are implementations of LaTeX packages. Packages described in the "*Blue Book*" (`makeidx`, `ifthen`, `graphics` —and `graphicx`!—, `color`, `alltt`) are provided. Additionally, quite a few extra packages are provided. I provide no full documentation for these packages, users should refer to the first pages of the package documentation, which can usually be found in the book [LaTeX-bis], in your local LaTeX installation or in a TeX CTAN-archive.

At the moment, most package options are ignored, except for the babel package, where it is essential.

B.17.1 AMS compatibility

HeVeA `amsmath` package defines some of the constructs of the `amsmath` package. At the moment, supported constructs are the `cases` environment and matrix environments [LaTeX-bis, Section 8.4], the environments for multi-line displayed equations (`gather`, `split`,...) [LaTeX-bis, Section 8.5] and the `\numberwithin` command [LaTeX-bis, Section 8.6.2].

HeVeA provides support for the `amssymb` symbols using Unicode. I found Unicode equivalent for most symbols. However, a few symbols remain undefined (*e.g.* `\varsubsetneqq`).

B.17.2 The `array` and `tabularx` packages

The `array` package is described in [LaTeX-bis, Section 5.3] and in the local documentation of modern LaTeX installations. It is a compatible extension of LaTeX arrays (see B.10.2). Basically, it provides new column specifications and a `\newcolumntype` construct for user-defined column specifications. Table 1 gives a summary of the new column specifications and of how HeVeA implements them.

Note that *centered*, *top-aligned* or *bottom-aligned* in the vertical direction, do not have exactly the same meaning in LaTeX and in HTML. However, the aspect is the same when all columns agree w.r.t. vertical

alignment. Ordinary column types (`c`, `l` and `r`) do not specify vertical alignment, which therefore becomes browser dependent.

The `>{`*decl*`}` and `<{`*decl*`}` constructs permit the encoding of TEX `\cases` macro as follows:

```
\def\cases#1{\left\{\begin{array}{l>{$}l<{$}}#1\end{array}\right.}
```

(This is an excerpt of the `latexcommon.hva` file.)

New column specifications are defined by the `\newcolumntype` construct:

`\newcolumntype{`*col*`}[`*narg*`]{`*body*`}`

Where *col* is one letter, the optional *narg* is a number (defaults to `0`), and `body` is built up with valid column specifications and macro-argument references (`#`*int*). Examples are:

```
\newcolumntype{C}{>{\bf}c}
\newcolumntype{E}[1]{*{#1}{c}}
\begin{tabular}{CE{3}}\hline
one & two & three & four \\
five & six & seven & eight \\ \hline
\end{tabular}
```

The column specification `C` means that entries will be typeset centered and using bold font, while the column specifications `E{`*num*`}` stands for *num* centered columns. We get:

one	two	three	four
five	six	seven	eight

HEVEA implements column specifications with commands defined in the `\newcommand` style. Thus, they have the same behaviour as regards double definition, which is not performed and induces a warning message. Thus, a column specification that is first defined in a `macro.hva` specific file, overrides the document definition.

The `tabularx` package [LATEX-bis, Section 5.3.5] provides a new tabular environment `tabularx` and a new column type `X`. HEVEA makes the former equivalent to `tabular` and the latter equivalent to `p{`*ignored*`}`. By contrast with the subtle array formatting that the `tabularx` package performs, this may seem a crude implementation. However, rendering is usually correct, although different.

More generally and from the HTML point of view such sophisticated formatting is browser job in the first place. However, the HTML definition allows suggested widths or heights for table entries and table themselves. From HEVEA point of view, drawing the border line between what can be specified and what can be left to the browser is not obvious at all. At the moment HEVEA choice is not to specify too much (in particular, all length arguments, either to column specifications or to the arrays themselves, are ignored). As a consequence, the final, browser viewed, aspect of arrays will usually be different from their printed aspect.

B.17.3 The `calc` package

The `calc` package enables using traditional, infix, notation for arithmetic operations inside the *num* argument to the `\setcounter{`*name*`}{`*num*`}` and `\addtocounter{`*name*`}{`*num*`}` constructs (see [LATEX-bis, Section A.4])

The `calc` package provides a similar extension of the syntax of the *len* argument to the `\setlength` and `\addtolength` constructs. HEVEA does not implement this extension, since it does not implement length registers in the first place.

B.17.4 Specifying the document input encoding, the `inputenc` package

The `inputenc` package enables LATEX to process a file according to various *8 bits* encodings, plus `UTF-8`. The one used encoding is specified as an option while loading the package `\usepackage[`*encoding*`]{inputenc}`. At the moment, HEVEA recognises ten latin encodings (from `latin1` to `latin10`), the `koi8-r` encoding, the

`ascii` encoding, four windows encodings, the `applemac` encoding, and the `utf8` encoding. It is important to notice that loading the `inputenc` package alters the HTML document charset. For instance if the `latin9` input encoding is selected by:

```
\usepackage[latin9]{inputenc}
```

Then, the document charset is `ISO-8859-15`, which is an enhanced version of `ISO-8859-1` with some characters for Œ, œ and €. The rationale behind changing the output document charset at the same time as changing the input encoding is to allow non-ascii bytes in the input file to be replicated as themselves in the output file.

However, one can change the document charset (and the output translator) by using the internal command `\@def@charset`. For instance, one can specify `latin1` encoding, while producing HTML pages in ascii:

```
\usepackage[latin1]{inputenc}
%HEVEA\@def@charset{US-ASCII}
```

See section 8.6 for a more thorough description of HTML charset management.

The `inputenc` package also provides the command `\inputcoding{`*encoding*`}` that changes the input encoding at any time. The argument *encoding* can be any of the options accepted by `\usepackage[`*encoding*`]{inputenc}`. The command `\inputcoding` of HeVeA follows the behaviour of its LaTeX counterpart, it the sense that it obeys scope rules. Notice that `\inputcoding` does not change the document output encoding and charset.

B.17.5 More symbols

HeVeA implements the following packages: `latexsym amssymb`, `textcomp` (a.k.a. "Text companion") and `eurosym` (a nice € symbol in LaTeX).

B.17.6 The `comment` package

The `comment` package provides two commands, `\excludecomment` and `\includecomment`, for (re-)defining new environments that ignore their content or that do nothing. The comment environment is also defined as an environment of the first kind.

B.17.7 Multiple Indexes with the `index` and `multind` packages

HeVeA supports several simultaneous indexes, following the scheme of the `index` package, which is present in modern LaTeX distributions. This scheme is backward compatible with the standard indexing scheme of LaTeX.

Support is not complete, but the most useful commands are available. More precisely, HeVeA knows the following commands:

`\newindex{`*tag*`}{`*ext*`}{`*ignored*`}{`*indexname*`}` Declare an index. The first argument *tag* is a tag to select this index in other commands; *ext* is the extension of the index information file generated by LaTeX (e.g., `idx`); *ignored* is ignored by HeVeA; and *indexname* is the title of the index. There also exists a `\renewindex` commands that takes the same arguments and that can be used to redefine previously declared indexes.

`\makeindex` Perform `\newindex{default}{idx}{ind}{Index}`.

`\index[`*tag*`]{`*arg*`}` Act as the LaTeX `\index` command except that the information extracted from *arg* goes to the *tag* index. The *tag* argument defaults to `default`, thereby yielding standard LaTeX behaviour for the `\index` command without an optional argument. There also exists a stared-variant `\index*` that Additionally typesets *arg*.

`\printindex[`*tag*`]` Compute, format and output index whose tag is *tag*. The *tag* argument defaults to `default`.

The `multind` package is supported to some extend, but `index` is definitely to be preferred.

B.17.8 "Natural" bibliographies, the `natbib` package

LaTeX version of `natbib`[3] is present in modern installations.

Implementation is quite complete and compatible with version 8.0 of the `natbib` package (with the `keyval` style command `\setcitestyle`).

Unimplemented features are the sorting and compression of references. Automatic generation of an index of citations is handled, but the current implementation probably is quite fragile.

B.17.9 Multiple bibliographies

The `multibib` package

HeVeA provides a slightly incomplete implementation of the `multibib` package. The one non-implemented feature is the simultaneous definition of more than one bibliography. That is one cannot invoke `\newcites` as follows:

```
\newcites{suf1, suf2}{Title1, Title2}
```

Instead, one should perform to calls to the `\newcites` command:

```
\newcites{suf1}{Title1}\newcites{suf2}{Title2}
```

The `chapterbib` package

A basic implementation is provided. At the moment, you can define one bibliography per included file and no toplevel bibliography. HeVeA implementation of this package recognises the option `sectionbib` and provides the command `\sectionbib` to change the sectioning command introduced by bibliographies.

B.17.10 Support for `babel`

B.17.10.1 Basics

HeVeA offers support for the LaTeX package `babel`. When it reads the command

```
\usepackage[lang-list]{babel}
```

it loads `babel.hva`, and sends it the saved `lang-list`. The file `babel.hva` then looks at each language (say `x`) in it, and loads `x.hva`, which offers support for the language `x`. As in LaTeX, the last language in the list is selected as default. As an example the command

```
\usepackage[english,french,german]{babel}
```

would load `babel.hva`, then the files `english.hva,french.hva,german.hva` containing the respective definitions, and finally activate the definitions in `german.hva` and sets the current language to german.

[3] `http://www.ctan.org/pkg/natbib.html`

B.17.10.2 Commands and languages

The following babel commands for changing and querying the language work as in LaTeX :

1. `\selectlanguage` : to change the language
2. `\iflanguage` : to branch after comparing with current language

The language specific details are described in the corresponding `.hva` file, just as in the `.sty` file for LaTeX. Users need to supply this file for their language, or modify/check the files if they are already supplied with the distribution. The list of languages is given below.

american	austrian	brazil	catalan
check	croatian	danish	dutch
english	esperanto	finnish	french
galician	german	italian	magyar
norsk	nynorsk	polish	portuguese
romanian	russian	slovak	slovene
spanish	swedish	turkish	

B.17.10.3 Writing `hva` files

The languages for which `.hva` files are available with the distribution are english, french, german, austrian, czech and portuguese. These may need to be modified as not all accents and hyphenation techniques are supported.

They can be written/modified as simple TeX files (see the section B.16.1.1 on writing TeX macros for details). As an example, one may also take a look at the file french.hva[4], which describes the details for french.

Note how all definitions are *inside* the definition for `\french@babel`, which is the command that `\selectlanguage{french` would call. Similar commands need to be provided (*i.e.* `\x@babel` in `\x.hva` for language `x`).

Notice that it is wise to write the `\x.hva` in plain ascii only. Some definitions may involve specifying Unicode characters, for doing so, using the `\@print@u` is recommended (cf. Section 8.3). The definition of Unicode characters can be found at http://www.unicode.org/charts/[5]. Most language specific Unicode characters can be found in the first few files.

B.17.11 The `url` package

LaTeX source[6].

This package in fact provides a enhanced `\verb` command that can appear inside other command arguments. This command is named `\url`, but it can be used for any verbatim text, including DOS-like path names. Hence, one can insert urls in one's document without worrying about LaTeX active characters:

```
This is a complicated url: \url{http://foo.com/~user#label%coucou}.
```

which gets typeset as: "This is a complicated url: `http://foo.com/~user#label%coucou`."

The main use for the `\url` command is to specify urls as arguments to HeVeA commands for hyperlinks (see section 8.1.1):

```
\hevea{} home page is
\ahrefurl{\url{http://hevea.inria.fr/}}
```

[4] `../html/french.hva`
[5] `http://www.unicode.org/charts/`
[6] `http://www.ctan.org/pkg/url.html`

It yields: "HeVeA home page is `http://hevea.inria.fr/`".

However the `\url` command is fragile, as a consequence it cannot be used inside `\footahref` first argument (This is a LaTeX problem, not an HeVeA one). The `url` package solves this problem by providing the `\urldef` command for defining commands whose body is typeset by using `\url`:

```
\urldef{\heveahome}{\url}{http://hevea.inria.fr/}
```

Such a source defines the robust command `\heveahome` as the intended url. Hence the following source works as expected:

```
Have a look at \footahref{\heveahome}{\hevea{} home page}
```

It yields: "Have a look at HeVeA home page[7]".

Using `\url` inside command definitions with a `#`i argument is a bad idea, since it gives "verbatim" a rather random meaning. Unfortunately, in some situations (e.g, no `%`, no `#`), it may work in LaTeX. By contrast, it does not work in HeVeA. In such situations, `\urldef` should be used.

HeVeA implementation is somehow compatible at the "programming level". Thus, users can define new commands whose argument is understood verbatim. The `urlhref.hva` style file from the distribution takes advantage of this to define the `\url` command, so that it both typesets an url and inserts a link to it. The `urlhref.hva` style file (which is an HeVeA style file and not a LaTeX style file) can be adequate for bibliographic references, which often use `\url` for its typesetting power. Of course, loading `urlhref.hva` only makes sense when all arguments to `\url` are urls...

B.17.12 Verbatim text: the `moreverb` and `verbatim` packages

These two packages provide new commands and environments for processing verbatim text. I recommend using `moreverb` rather than `verbatim`, since HeVeA implementation is more advanced for the former package.

B.17.13 Typesetting computer languages: the `listings` package

I strongly recommend the `listings` package. Learning the user interface requires a little effort, but it is worth it.

HeVeA features a quite compatible implementation, please refer to the original package documentation. Do not hesitate to report discrepancies. Note that HeVeA does not produce very compact HTML in case you use this package. This can be cured by giving `hevea` the command-line option `-O` (see C.1.1.4).

The `lstlisting` environment is styled through an homonymous style class (see 9.2 and 9.3) and most `lstlisting` environments get translated to `div` elements with the appropriate `\getenvclass{lstlisting}` class, which, by default is `lstlisting`. A few points deserve mention:

1. The definition of default style class `lstlisting` includes the important declarations `font-family:monospace;` and `white-space:pre;`, which, more or less, specify non-proportional font and mandatory line breaks. In case you replace `lstlisting` by another style class (by `\setenvclass{lstlisting}{`*another one*`}`), your alternate definition should probably feature an identical specification. Otherwise, rendering would be poor, as regards spacing and line breaks.

2. When listings are framed, that is, when some `frame=...` or `background=...` keyval specifications are active, they no longer get translated to `div` elements. Instead they get translated to one cell tables whose `td` and `table` elements are styled through style classes `lstlisting` and `lstframe`, respectively. Of course, those two style classes follow the usual `\setenvclass`/`\getenvclass` mechanism. That way, one can for instance center all framed listings by issuing the following declaration in the document preamble:

   ```
   \newstyle{.lstframe}{margin:auto;}
   ```

[7] `http://hevea.inria.fr/`

Notice that the default style class `lstframe` is empty.

3. Unfortunately the `white-space:pre;` style declaration is still a bit young, and some browsers implement it in rather incomplete fashion. This is particularly true as regards text copy-pasted from browser display. In case you want to provide your readers with easy copy-paste of `listings`, you can, by issuing the command `\lstavoidwhitepre` in the document preamble. Then, `white-space:pre;` is not used any longer: spaces get rendered by non-breaking space entities and line-breaks by `<BR>` elements, which significantly increase output size. However, as a positive consequence, display remains correct and text copy-pasted from browser display indeed possesses the line-breaks shown in display.

B.17.14 (Non-)Multi page tabular material

Those two packages provide LaTeX users with the possibility to typeset tabular material over several pages [LaTeX-bis, Section 5.4]. Of course, HeVeA does not care much about physical pages. Thus the `supertabular` and `longtable` environments are rendered more or less as `tabular` environments inside `table` environments.

B.17.15 Typesetting inference rules: the `mathpartir` package

The `mathpartir` package, authored by D. Rémy, essentially provides two features:

1. An environment `mathpar` for typesetting a sequence of math formulas in mixed horizontal and vertical mode. The environment selects the best arrangement according to the line width, exactly as paragraph mode does for words.

2. A command `\inferrule` (and its starred variant) for typesetting inferences rules.

We give a short description, focussing on HeVeA-related details. Users are encouraged to refer to the original documentation[8] of the package.

In the following, comments on rule typesetting apply to HeVeA output and not to LaTeX output.

B.17.15.1 The mathpar environment

In its LaTeX version, the `mathpar` environment is a "paragraph mode for formulas". It allows to typeset long list of formulas putting as many as possible on the same line:

```
\begin{mathpar}
A-Formula \and
Longer-Formula \and
And \and The-Last-One
\end{mathpar}
```

$$A-Formula \qquad Longer-Formula$$
$$And \qquad The-Last-One$$

In the example above, formulas are separated with `\and`. The LaTeX implementation also changes the meaning of paragraph breaks (either explicit as a `\par` command or implicit as a blank line) to act as `\and`. It also redefines the command `\\` as an explicit line-break in the flow of formulas.

```
\begin{mathpar}
\int_0^2 xdx = \frac{3}{2}
\\
\int_0^3 xdx = \frac{5}{2}
\end{mathpar}
```

$$\int_0^2 xdx = \frac{3}{2}$$
$$\int_0^3 xdx = \frac{5}{2}$$

[8] `http://pauillac.inria.fr/~remy/latex/index.html#tir`

The HeVeA version is simplistic: Formulas are typeset in math display mode, `\and` separators always produce horizontal space, while \\ always produce line-breaks. However, when prefixed by `\hva` the meaning of explicit separators is reversed: that is, `\hva\and` produces a line-break, while `\hva\\` produces horizontal space. Hence, we can typeset the previous example on two lines:

```
\begin{mathpar}
A-Formula \and
Longer-Formula \hva\and
And \and The-Last-One
\end{mathpar}
```

$$A-Formula \qquad Longer-Formula$$
$$And \qquad The-Last-One$$

It is to be noticed that the LaTeX version of the package defines `\hva` as a no-op, so as to allow explicit instructions given to HeVeA not to impact on the automatic typesetting performed by LaTeX.

B.17.15.2 The inferrule macro

The `\inferrule` macro is designed to typeset inference rules. It should only be used in math mode (or display math mode). It takes three arguments, the first being optional, specifying the label, premises, and conclusions respectively. The premises and the conclusions are both lists of formulas, and are separated by \\. A simple example of its use is

```
\inferrule
  [label]
  {one \\ two \\ three \\ or \\ more \\ premises}
  {and \\ any \\ number \\ of \\ conclusions \\ as \\ well}
```

which gives the following rendering:

$$\text{LABEL}\ \frac{one \quad two \quad three \quad or \quad more \quad premises}{and \quad any \quad number \quad of \quad conclusions \quad as \quad well}$$

Again, HeVeA is simplistic. Where LaTeX performs actual typesetting, interpreting \\ as horizontal or vertical breaks, HeVeA always interpret \\ as an horizontal break. In fact HeVeA interpret all separators (\\, `\and`) as horizontal breaks, when they appear in the arguments of the `\inferrule` command. Nevertheless prefixing separators with `\hva` yields vertical breaks:

```
\inferrule
   {aa \hva\\ bb}
   {dd \\ ee \\ ff}
```

$$\frac{aa \quad bb}{dd \quad ee \quad ff}$$

The color of the horizontal rule that separates the premises and conclusions can be changed by redefining the command `\mpr@hhline@color`. This color must be specified as a low-level color (cf. Section B.14.2.2).

B.17.15.3 Options

By default, lines are centered in inference rules. However, this can be changed either by using `\mprset{flushleft}` or `\mprset{center}`, as shown below.

```
$$\mprset{flushleft}
  \inferrule
    {a \\ bbb \hva\\ ccc \\ dddd}
    {e \\ ff \hva\\ gg}
$$
```

$$\mprset{flushleft}\inferrule{a \\ bbb \hva\\ ccc \\ dddd}{e \\ ff \hva\\ gg}$$

B.17.15.4 Derivation trees

The `mathpartir` package provides a starred variant `\inferrule*`. In LaTeX, the boxes produced by `\inferrule` and `\inferrule*` differ as regards their baseline, the second being well adapted to derivation trees. All this is irrelevant to HeVeA, but `\inferrule*` remains of interest because of its interface: the optional argument to the `\inferrule*` command is a list of *key=value* pairs in the style of `keyval`. This makes the variant command much more flexible.

key	Effect for value *v*
`before`	Execute *v* before typesetting the rule. Useful for instance to change the maximal width of the rule.
`left`	Put a label *v* on the left of the rule
`Left`	Idem.
`right`	As `left`, but on the right of the rule.
`Right`	As `Left`, but on the right of the rule.
`lab`	Put a label *v* above the inference rule, in the style of `\inferrule`.
`Lab`	Idem.
`vdots`	Raise the rule by *v* and insert vertical dots, the length argument is translated to a number of line-skips.

Additionally, the value-less key `center` centers premises and conclusions (this is the default), while `flushleft` commands left alignment of premises and conclusions (as `\mprset{flushleft}` does). Other keys defined by the LaTeX package exist and are parsed, but they perform no operation.

As an example, the code

```
\begin{mathpar}
\inferrule* [Left=Foo]
   {\inferrule* [Right=Bar,width=8em,
                 leftskip=2em,rightskip=2em,vdots=1.5em]
      {a \and a \and bb \hva\\ cc \and dd}
      {ee}
    \and ff \and gg}
   {hh}
\hva\and
\inferrule* [lab=XX]{uu \and vv}{ww}
\end{mathpar}
```

produces the following output:

$$\dfrac{\dfrac{a \quad a \quad bb}{cc \quad dd}\,\textsc{Bar}}{\textsc{Foo}\ \dfrac{\vdots}{ee}\quad ff \quad gg}{hh} \qquad \dfrac{\text{XX}}{uu \quad vv}{ww}$$

B.17.16 The `ifpdf` package

This package should be present in modern `latex` installations. Basically, the package defines a boolean register `pdf`, whose value is true for tools that produce PDF (such as `pdflatex`) and false for tools that produce DVI (such as `latex`).

The hevea version of the package simply defines the boolean register `pdf` with initial value true. Command-line option `-pdf` is also added to `imagen` command-line options (by using the command `\@addimagenopt`, see Section 10.7). As a result, `imagen` will normally call `pdflatex` in place of `latex`.

In case standard `latex` processing in `imagen` is wished, one can issue the command `\pdffalse` after loading the ifpdf package and before `\begin{document}`. Then, no command line option is added. Hence, to achieve `latex` processing of the *image* file, while still loading the `ifpdf` package, one writes:

```
\usepackage{ifpdf}
%HEVEA\pdffalse
```

B.17.17 Typesetting Thai

HeVeA features an implementation of Andrew Seagar's technique for Thai in LaTeX, by the means of the package `thai.hva` in the distribution.

As regards input encoding, Thai users of HeVeA could (perhaps) use `\usepackage[utf8]{inputenc}`. However, the typesetting of Thai is more subtle than just proper characters. For that reason, Thai in LaTeX is better performed by another technique, which HeVeA supports. See this specific document[9].

B.17.18 Hanging paragraphs

The `hanging` package is implemented. HeVeA implementation consists of no-ops, except for the `hangparas` environment, which is partially implemented. Assume the following usage of `hangparas`:

`\begin{hangparas}{`*wd*`}{`*n*`}` ... `\end{hangparas}`

where *wd* is a length that makes sense both for LaTeX and CSS (typically `2ex`). Then HTML output will reproduce LaTeX output for $n = 1$, regardless of the given value of argument n. That is, in any paragraph inside the environment, all lines except the first get indented by *wd*.

B.17.19 The `cleveref` package

The `cleveref` package attempts (and mostly succeeds) typesetting references cleverly. Its main feature is a `\cref` command that accepts several, comma separated, label references and typesets them as a list (which can be one-element long, of course) prefixed with sectional unit names. The HeVeA implementation is quite complete, but it does not support some of the subtleties of the LaTeX implementations, especially as regards customisation.

[9] `http://hevea.inria.fr/doc/thaihevea.html`

B.17.20 Other packages

The `fancyverb` and `colortbl` packages are partly implemented.

The `xspace` package is implemented, in simple cases, rendering is satisfactory, but beware: HeVeA differs significantly from TeX, and discrepancies are likely.

The `chngcntr` package is implemented. This package provides commands to connect (and disconnect) counters once they are created.

The `import` package is partially implemented: all starred commands are missing.

The `booktabs` package is implemented. This package provides nicer rulers in tables as specific commands. HeVeA defines those as no-ops.

Part C
Practical information

C.1 Usage

C.1.1 HeVeA usage

The `hevea` command has two operating modes, normal mode and filter mode. Operating mode is determined by the nature of the last command-line argument.

C.1.1.1 Command line arguments

The `hevea` command interprets its arguments as names of files and attempts to process them. Given an argument *filename* there are two cases:

- If *filename* is *base*`.tex` or *base*`.hva`, then a single attempt to open *filename* is made.
- In other cases, a first attempt to open *filename*`.tex` is made. In case of failure, a second attempt to open *filename* is made.

In all attempts, implicit filenames are searched along `hevea` search path, which consist in:

1. the current directory "`.`",
2. user-specified directories (with the `-I` command-line option),
3. `hevea` library directory.
4. one of the sub-directories `html`, `text` or `info` from `hevea` library directory, depending upon `hevea` output format,

The `hevea` library directory is fixed at compile-time (this is where `hevea` library files are installed) and typically is `/usr/local/lib/hevea`. However, this compile-time value can be overridden by setting the `HEVEADIR` shell environment variable. In all cases, the value of `hevea` library directory can be accessed from the processed document as the value of the command `\@hevealibdir`.

C.1.1.2 Normal mode

If the last argument has an extension that is different from `.hva` or has no extension, then it is interpreted as the name of the *main input file*. The main input file is the document to be translated and normally contains the `\documentclass` command. In that case two *basenames* are defined:

- The input basename, *basein*, is defined as the main input file name, with extension removed when present.
- The output basename, *baseout*, is *basein* with leading directories omitted. However the output basename can be changed, using the `-o` option (see below).

HeVeA will attempt to load the main input file. Ancillary files from a previous run of LaTeX (*i.e.* `.aux`, `.bll` and `.idx` files) will be searched as *basein*`.`*ext*. The output base name governs all files produced by HeVeA. That is, HTML output of HeVeA normally goes to the file *baseout*`.html`, while cross-referencing information goes into *baseout*`.haux`. Furthemore, if an *image* file is generated (cf. section 6), its name will be *baseout*`.image.tex`.

Thus, in the simple case where the `hevea` command is invoked as:

```
# hevea file.tex
```

The input basename is `file` and the output basename also is `file`. The main input file is searched once along `hevea` search path as `file.tex`. HTML output goes into file `file.html`, in the current directory. In the more complicated case where the `hevea` command is invoked as:

```
# hevea ./dir/file
```

The input base name is `./dir/file` and the output basename is `file`. The main input file is loaded by first attempting to open file `./dir/file.tex`, then file `./dir/file`. HTML output goes into file `file.html`, in the current directory.

Finally, the output base name can be a full path, but you have to use option `-o`. For instance, we consider:

```
# hevea -o out/out.html file.tex
```

Then, HTML output goes into `out/out.html` — notice that directory `out` must exist. Furthermore, `hevea` output base name is `out/out`. This means that `hevea` generates files `out/out.haux`, `out/out.image.tex` etc.

The `article.hva`, `seminar.hva`, `book.hva` and `report.hva` base style files from HeVeA library are special. Only the first base style file is loaded and the `\documentclass` command has no effect when a base style file is already loaded. This feature allows to override the document base style. Thus, a document `file.tex` can be translated using the *article* base style as follows:

```
# hevea article.hva file.tex
```

C.1.1.3 Filter mode

If there is no command-line argument, or if the last command-line argument has the extension `.hva`, then there is neither input base name nor output base name, the standard input is read and output normally goes to the standard output. Output starts immediately, whithout waiting for `\begin{document}`. In other words `hevea` acts as a filter.

Please note that this operating mode is just for translating isolated LaTeX constructs. The normal way to translate a full document *file*`.tex` being "`hevea` *file*`.tex`" and not "`hevea <` *file*`.tex >` *file*`.html`".

C.1.1.4 Options

The `hevea` command recognises the following options:

`-version` Show `hevea` version and exit.

`-v` Verbose flag, can be repeated to increase verbosity. However, this is mostly for debug.

`-dv` Add border around some of the block-level elements issued. Specifically, all `div` and `p` are bordered, while the structure of displayed material is also shown.

`-s` Suppress warnings.

`-I` *dirname* Add *dirname* to the search path.

`-o` *name* Make *name* the output basename. However, if *name* is *base*`.html`, then the output basename is *base*. Besides, `-o -` makes HEVEA output to standard output.

`-e` *filename* Prevent `hevea` from loading any file whose name is *filename*. Note that this option applies to all files, including `hevea.hva` and base style files.

`-fix` Iterate HEVEA until a fixpoint is found. Additionally, images get generated automatically.

`-O` Optimise HTML by calling `esponja` (see section C.1.3).

`-exec` *prog* Execute file *prog* and read the output. The file *prog* must have execution permission and is searched by following the searching rules of `hevea`.

`-francais` Deprecated by `babel` support. This option issues a warning message.

`-help` Print version number and a short help message.

The following options control the HTML code produced by `hevea`. By default, `hevea` outputs a page encoded in US-ASCII with most symbols rendered as HTML or numerical Unicode entities.

`-entities` Render symbols by using entities. This is the default.

`-textsymbols` Render symbols by English text.

`-moreenties` Enable the output of some infrequent entities. Use this option to target browsers with wide entities support.

`-mathml` Produces MathML output for equations, very experimental.

`-pedantic` Be strict in interpreting HTML definition. In particular, this option disable size and color changes inside `<PRE>`... `</PRE>`, which are otherwise performed.

The following options select and control alternative output formats (see section 11):

`-text` Output plain text. Output file extension is `.txt`.

`-info` Output info format. Output file extension is `.info`.

`-w` *width* Set the line width for text or info output, defaults to 72.

Part A of this document is a tutorial introduction to HEVEA, while Part B is the reference manual of HEVEA.

C.1.2 HACHA usage

The `hacha` command interprets its argument *base*`.html` as the name of a HTML source file to cut into pieces. It also recognises the following options:

`-v` Be a little verbose.

`-o` *filename* Make HACHA output go into file *filename* (defaults to index.html). Additionally, if *filename* is a composite filename, *dir/base*, then all files outputted by HACHA will reside in directory *dir*.

`-tocbis` Another style for table of contents: sub-tables are replicated at the beginning of every file.

`-tocter` Like `-tocbis` above, except that sub-tables do not appear in the main table of contents (see Section 7.2.3).

`-no-svg-arrows` Use GIF arrows for the Previous/Up/Next links, in place of the default SVG arrows.

`-nolinks` Do not insert Previous/Up/Next links in generated pages.

`-hrf` Output a *base*`.hrf` file, showing in which output files are the anchors from the input file gone. The format of this summary is one "*anchor*`\t`*file*" line per anchor. This information may be needed by other tools.

`-help` Print version number and a short help message.

Section 7 of the user manual explains how to alter HACHA default behaviour.

C.1.3 `esponja` usage

The program `esponja` is part of HEVEA and is designed to optimise `hevea` output. However, `esponja` can also be used alone to optimise text-level elements in HTML files. Since `esponja` fails to operate when it detects incorrect HTML, it can be used as a partial HTML validator.

C.1.3.1 Operating mode

The program `esponja` interprets its arguments as names of files and attempt to process them. It is important to notice that `esponja` will *replace* files by their optimised versions, unless instructed not to do so with option `-n`.

Invoking `esponja` as

```
# esponja foo.html
```

will alter `foo.html`. Of course, if `esponja` does not succeed in making `foo.html` any smaller or if `esponja` fails, the original `foo.html` is left unchanged. Note that this feature allows to optimise all HTML files in a given directory by:

```
# esponja *.html
```

C.1.3.2 Options

The command `esponja` recognises the following options:

`-v` Be verbose, can be repeated to increase verbosity.

`-n` Do not alter input files. Instead, `esponja` output for file *input*`.html` goes to file *input*`.esp`. Option `-n` implies option `-v`.

`-u` Output `esponja` intermediate version of HTML. In most occasions, this amounts to pessimize instead of to optimise. It may yield challenging input for other HTML optimisers.

C.1.4 `bibhva` usage

The program `bibhva` is a simple wrapper, which basically forces `bibtex` into accepting a `.haux` file as input and producing a `.hbbl` file as output. Usage is `bibhva` *bibtex-options basename*.

C.1.5 `imagen` usage

The command `imagen` is a simple shell script that translates a LaTeX document into many `.png` images. The `imagen` script relies on much software to be installed on your computer, see Section C.4.1.

It is a companion program of HeVeA, which must have been previously run as:

```
# hevea... base.tex
```
or
```
# hevea... -o base.html...
```

In both cases, *base* is HeVeA output basename. When told to do so (see section 6) HeVeA echoes part of its input into the *base*`.image.tex` file.

The `imagen` script should then be run as:

```
# imagen base
```

The `imagen` script produces one *base*`nnn.png` image file per page in the *base*`.image.tex` file.

This is done by first calling `latex` on *base*`.image.tex`, yielding one `dvi` file. Then, `dvips` translates this file into one single Postscript file that contains all the images, or into one Postscript file per image, depending upon your version of `dvips`. Postscript files are interpreted by ghostscript (`gs`) that outputs `ppm` images, which are then fed into a series of transformations that change them into `.png` files.

The `imagen` script recognises the following options:

`-mag` *nnnn* Change the enlarging ratio that is applied while translating DVI into Postscript. More precisely, `dvips` is run with `-x`*nnnn* option. Default value for this ration is 1414, this means that, by default, `imagen` magnifies LaTeX output by a factor of 1.414.

`-extra` *command* Insert *command* as an additional stage in `imagen` `ppm` to `png` production chain. *command* is an Unix filter that expects a `ppm` image in its standard input and outputs a `ppm` image on its standard output. A sensible choice for *command* is one command from the `netpbm` package, several such commands piped together, or various invocations of the `convert` command from **ImageMagick**.

`-quant` *number* Add an extra color quantisation step in `imagen` `ppm` image production chain, where *number* is the maximal number of colors in the produced images. This option may be needed as a response to a failure in the image production chain. It can also help in limiting image files size.

`-png` Output PNG images. This is the default.

`-gif` Output GIF images in place of PNG images. GIF image files have a `.gif` extension. Note that `hevea` should have been previously run as `hevea gif.hva` *base*`.tex` (so that the proper `.gif` filename extension is given to image file references from within the HTML document).

`-svg` Output SVG images in addition to PNG (or GIF) images. Note that `hevea` should have been previously run as `hevea svg.hva` *base*`.tex`.

`-pnm` Output PPM images. This option mostly serves debugging purposes. Experimented users can also take advantage of it for performing additional image transformation or adopting exotic image formats.

`-t` *arg* Pass option "`-t` *arg*" to `dvips`. For instance, using "`-t a3`" may help when images are truncated on the right.

`-pdf` Have `imagen` call `pdflatex` instead of `latex`.

The first three options enable users to correct some misbehaviours. For instance, when the document base style is *seminar*, image orientation may be wrong and the images are too small. This can be cured by invoking `imagen` as:

```
# imagen -extra "pnmflip -ccw" -mag 2000 base
```

Notice that hevea calls imagen by itself, when given the command-line option -fix. In that situation, the command-line options of imagen can be controlled from source file by using the command \@addimagenopt (see Section 10.7).

C.1.6 Invoking hevea, hacha and imagen

In this section, we give a few sequence of (Unix) commands to build the HTML version of a document in various situations. The next section gives a few Makefile's for similar situations.

We translate a file doc.tex that requires a specific style file doc.hva. The file is first translated into doc.html by hevea, which also reads the specific style file doc.hva. Then, hacha cuts doc.html into several, doc001.html, doc002.html, etc. also producing the table of links file index.html.

```
# hevea -fix doc.hva doc.tex
# hacha doc.html
```

Thanks to the command-line option -fix, hevea runs the appropriate number of times automatically. In case hevea produces a non-empty doc.image.tex file, then hevea calls imagen by itself (because of option -fix). Hence, the above sequence of two commands is also appropriate in that situation.

In case some problem occurs, it is sometime convenient to run imagen by hand. It is time *not* to use the option -fix.

```
# rm -f doc.image.tex
# hevea doc.hva doc.tex
```

Now, hevea normally has shown the imagen command line that it would have run, if it had been given the option -fix. For instance, if doc.hva includes \input{gif.hva}, then hevea shows the following warning:

```
HeVeA Warning: images may have changed, run 'imagen -gif doc'
```

Now, one can run imagen as it should be.

It is sometime convenient not to clobber the source directory with numerous target files. It suffices to instruct hevea and hacha to output files in a specific directory, say doc.

```
# hevea -fix -o doc/doc.html doc.hva doc.tex
# hacha -o doc/index.html doc/doc.html
```

Notice that hevea does not create the target directory doc: it must exist before hevea runs. Again, in case hevea calls imagen, image generation should proceed smoothly and the final files doc001.png, doc002.png, ... should go into directory doc.

In all situations, while installing files to their final destination, it is important not to forget any relevant files. In particular, in addition to the root file (index.html), contents files (doc001.html, doc002.html, etc.) and images (doc001.png, doc002.png, etc.), one should not forget the arrow images and the style sheet generated by hacha (contents_motif.gif, next_motif.gif, previous_motif.gif and doc.css).

As a consequence, producing all files into the subdirectory doc may be a good idea, since then one easily install all relevant files by copying doc to a public destination.

```
# cp -r doc $(HOME)/public_html
```

However, one then also installs the auxiliary files of hevea, and hevea output file doc.html, which is no longer useful once hacha has run. Hence, those should be erased beforehand.

```
# rm -f doc/doc.h{tml,aux,ind,toc} doc/doc.image.tex
# cp -r doc $(HOME)/public_html
```

C.1.7 Using `make`

Here is a typical `Makefile`, which is appropriate when no images are produced.

```
HEVEA=hevea
HEVEAOPTS=-fix
HACHA=hacha
#document base name
DOC=doc
index.html: $(DOC).html
        $(HACHA) -o index.html $(DOC).html

$(DOC).html: $(DOC).hva $(DOC).tex
        $(HEVEA) $(HEVEAOPTS) $(DOC).hva $(DOC).tex

clean:
        rm -f $(DOC).html $(DOC).h{toc,aux,ind}
        rm -f index.html $(DOC)[0-9][0-9][0-9].html $(DOC).css
```

Note that the `clean` rule removes all the `doc001.html`, `doc002.html`, etc. and `doc.css` files produced by `hacha`. Also note that `make clean` also removes the `doc.haux`, `doc.hind` and `doc.htoc` files, which are HeVeA auxiliary files.

When the *image* file feature is used, one can use the following, extended, `Makefile`:

```
HEVEA=hevea
HEVEAOPTS=-fix
HACHA=hacha
#document base name
DOC=doc
index.html: $(DOC).html
        $(HACHA) -o index.html $(DOC).html

$(DOC).html: $(DOC).hva $(DOC).tex
        $(HEVEA) $(HEVEAOPTS) $(DOC).hva $(DOC).tex

clean:
        rm -f $(DOC).html $(DOC).h{toc,aux,ind}
        rm -f index.html $(DOC)[0-9][0-9][0-9].html $(DOC).css
        rm -f $(DOC).image.* $(DOC)[0-9][0-9][0-9].png *_motif.gif
```

Observe that the `clean` rule now also gets rid of `doc.image.tex` and of the various files produced by `imagen`.

With the following `Makefile`, `hevea`, `imagen`, `hacha` all output their files into a specific directory `DIR`.

```
HEVEA=hevea
HEVEAOPTS=-fix
HACHA=hacha
#document base name
DOC=doc
DIR=$(HOME)/public_html/$(DOC)
BASE=$(DIR)/$(DOC)

$(DIR)/index.html: $(BASE).html
        $(HACHA) -tocter -o $(DIR)/index.html $(BASE).html
```

```
$(BASE).html: $(DOC).hva $(DOC).tex
        $(HEVEA) $(HEVEAOPTS) $(DOC).hva -o $(BASE).html $(DOC).tex

partialclean:
        rm -f $(BASE).h{tml,aux,toc,ind} $(BASE).image.*

clean:
        rm -f $(DIR)/*
```

The above `Makefile` directly produces HTML and PNG files into the final directory `$(HOME)/public_html/$(DOC)`. The new `partialclean` entry erases files that are not useful anymore, once `imagen` and `hacha` have performed their tasks.

However, most often, it is more appropriate to build HTML and PNG files in a specific directory, and then to copy them to their final destination.

```
   ...

#document base name
DOC=doc
DIR=$(DOC)
BASE=$(DIR)/$(DOC)
INSTALLDIR=$(HOME)/public_html/$(DOC)

  ...

install: partialclean
        cp $(DIR)/* $(INSTALLDIR)
  ...
```

C.2 Browser configuration

By default, HeVeA does not anymore use the `FACE=symbol` attribute to the `<FONT ...>` tag. As a consequence, browser configuration is no longer needed.

HeVeA now extensively outputs Unicode entities. This first means that HeVeA targets modern browsers with decent unicode support, and only those.

In case your browser is recent and that you nevertheless experience display problems on HeVeA-generated pages, see the excellent Alan Wood's Unicode Resources[10]. It may help to understand display problems and even to solve them by configuring browsers or installing some fonts.

C.3 Availability

C.3.1 Internet stuff

HeVeA home page is `http://hevea.inria.fr/`. It contains links to the on-line manual[11] and to the distribution[12].

The author can be contacted at `Luc.Maranget@inria.fr`.

[10] `http://www.alanwood.net/unicode/`
[11] `http://hevea.inria.fr//doc/`
[12] `http://hevea.inria.fr/distri`

C.3.2 Law

HeVeA can be freely used and redistributed without modifications. Modifying and redistributing HeVeA implies a few constraints. More precisely, HeVeA is distributed under the terms of the Q Public License, but HeVeA binaries include the Objective Caml runtime system, which is distributed under the Gnu Library General Public License (LGPL). See the LICENSE[13] file for details.

The manual itself is distributed under the terms of the Free Document Dissemination Licence[14].

C.4 Installation

C.4.1 Requirements

The programs `hevea` and `hacha` are written in Objective Caml[15]. Thus, you really need Objective Caml (the more recent version, the better) to compile them. However, some binary distributions exist, which are managed by people other than me (thanks to them). Links to some of these distributions appear in HeVeA home page.

HeVeA users may instruct the program not to process a part of the input (see section 6). Instead, this part is processed into a bitmap file and HeVeA outputs a link to the image file. LaTeX source is changed into `.png` images by the `imagen` script, which basically calls, LaTeX, `dvips`, `ghostscript`[16] and the `convert` command from the image processing package **ImageMagick**[17].

To benefit from the full functionality of HeVeA, you need all this software. However, HeVeA runs without them, but then you will have to produce images by yourself.

C.4.2 Principles

The details are given in the `README` file from the distribution. Basically, HeVeA should be given a library directory. The installation procedure stores the `hevea.hva` and base style files in this directory. There are two compilation modes, the `opt` mode selects the native code OCaml compiler `ocamlopt`, while the `byte` mode selects the bytecode OCaml compiler `ocamlc`. In HeVeA case, `ocamlopt` produces code that is up to three times as fast as the one produced by `ocamlc`. Thus, default compilation mode is `opt`, however it may be the case on some systems that only `ocamlc` is available.

Note that, when installing HeVeA from the source distribution, the `hevea.sty` (and `mathjax.sty`) style files are simply copied to HeVeA library directory. It remains users (and package maintainers) responsibility to make those files accessible to LaTeX.

C.5 Other LaTeX to HTML translators

This short section gives pointers to a few other translators. I performed not extensive testing and make no thorough comparison.

LaTeX2html LaTeX2html is a full system. It is written in perl and calls LaTeX when in trouble. As a consequence, LaTeX2html is powerful but it may fail on large documents, for speed and memory reasons. More information on LaTeX2html can be found at

`http://www.latex2html.org/`

[13] `http://hevea.inria.fr/distri/LICENSE`
[14] `http://pauillac.inria.fr/~lang/licence/v1/fddl.html`
[15] `http://caml.inria.fr/ocaml/`
[16] `http://www.cs.wisc.edu/~ghost/index.html`
[17] `http://www.imagemagick.org/`

TTH The principle behind TTH is the same as the one of HeVeA: write a fast translator as a lexer, use symbol fonts and tables. However, there are differences, TTH accepts both TeX and LaTeX source, TTH is written in C and the full source is not available (only `lex` output is available). Additionally, TTH insist on not using any kind of LaTeX generated information and will show proper cross-reference labels, even when no `.aux` file is present. TTH output is a single document, whereas HaCha can cut the output of HeVeA into several files. (however there exists a commercial version of TTH that provides this extra functionality). TTH can be found at

`http://hutchinson.belmont.ma.us/tth/`.

`TeX4ht` TeX4ht is a highly configurable TeX-based authoring system dedicated mainly to produce hypertext. It interacts with TeX-based applications through style files and postprocessors, leaving the processing of the source files to the native TeX compiler. As a result, TeX4ht may be more powerful than HeVeA, but may also be more difficult to configure. More information on TeX4ht can be found at:

`http://www.tug.org/tex4ht/`

`htmlgen` The `htmlgen` translator is specialized for producing the Caml manuals. This is HeVeA direct ancestor and I owe much to its author, X. Leroy. See [htmlgen] for a description of `htmlgen` and a (bit outdated) discussion on LaTeX to HTML translation.

C.6 Acknowledgements

The following people contributed to HeVeA development:

- Philip A.Viton, maintains a Windows (win32) port of HeVeA.
- Tibault Suzanne authored the HTML 5 generator that now is the default generator of HeVeA.
- Abhishek Thakur implemented most of the new features of version 1.08, including, translations of symbols to Unicode entities, the `babel` package, and style sheet support.
- Christian Queinnec wrote an extra lexer to translate code snippets produced by its tool VideoC for writing pedagogical documents on programming. The very principle he introduced for interfacing the `videoc` lexer with HeVeA main lexer is now used extensively throughout HeVeA source code.
- Andrew Seagar is at the origin of support for the Thai language. He is the author of the document "How to Use HeVeA with the Thai Character Set".
- Pierre Boulet, by using HeVeA as a stage in his tool MlDoc for documenting Objective Caml source code, forced me into debugging HeVeA implementation of the `alltt` environment.
- Nicolas Tessaud implemented the `-text` and `-info` output modes (see section 11).
- Georges Mariano asked for many feature, and argued a lot to have them implemented.
- Many users contributed by sending bug reports.

References

[LaTeX-bis] M. Gooseens, F. Mittelbach, A. Samarin. *The LaTeX Companion* Addison-Websley, 1994.

[LaTeX] L. Lamport. *A Document Preparation System System, LaTeX, User's Guide and Reference Manual.* Addison-Websley, 1994.

[htmlgen] X. Leroy. *Lessons learned from the translation of documentation from LaTeX to HTML*. ERCIM/W4G Int. Workshop on WWW Authoring and Integration Tools, 1995. Available on the web at `http://cristal.inria.fr/~xleroy/w4g.html`

[HTML-4.0] D. Ragget, A. Le Hors and I. Jacobs. *HTML 4.0 Reference Specification*. Available on the web at `http://www.w3.org/TR/REC-html40`, 1997.

[HTML-5a] W3C HTML Working groups. *HTML5 A vocabulary and associated APIs for HTML and XHTML* `http://dev.w3.org/html5/spec/spec.html`, 2012.

[HTML-5b] *HTML Living Standard* `http://www.whatwg.org/specs/web-apps/current-work/multipage/`, 2012.

[CSS-2] Bert Bos, Tantek Çelik, Ian Hickson and Håkon Wium Lie. *Cascading Style Sheets, Level 2 Revision 2 Specification*. Available on the web at `http://www.w3.org/TR/REC-CSS2/`, 2011.

Index