

# Melt User Manual

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	What is Melt? . . . . .	2
1.2	Getting Started . . . . .	2
<b>2</b>	<b>The Melt Distribution</b>	<b>4</b>
2.1	Download . . . . .	5
2.2	Pre-requisites . . . . .	5
2.3	Configure . . . . .	5
2.4	Compile . . . . .	5
2.5	Install . . . . .	5
2.6	Uninstall . . . . .	5
<b>3</b>	<b>The Latex Library</b>	<b>6</b>
3.1	Using the Library . . . . .	6
3.1.1	Compiling the Library . . . . .	6
3.1.2	Installing the Library . . . . .	6
3.1.3	Compiling a Program Which Uses the Library . . . . .	6
3.1.4	OcamlDoc Documentation . . . . .	6
3.2	The Kernel . . . . .	6
3.2.1	Raw Text . . . . .	7
3.2.2	Modes . . . . .	7
3.2.3	Concatenation . . . . .	7
3.2.4	Commands and Environments . . . . .	7
3.2.5	Printing the AST . . . . .	8
3.3	Pervasive Definitions . . . . .	9
3.3.1	References and Labels . . . . .	9
3.3.2	Verbatim . . . . .	9
<b>4</b>	<b>The Melt Preprocessor</b>	<b>12</b>
<b>5</b>	<b>The Melt Library</b>	<b>12</b>
5.1	Using the Library . . . . .	12
5.1.1	Compiling the Library . . . . .	12
5.1.2	Installing the Library . . . . .	13
5.1.3	Compiling a Program Which Uses the Library . . . . .	13
5.1.4	OcamlDoc Documentation . . . . .	13
5.2	..... . . . .	13
<b>6</b>	<b>The Melt Tool</b>	<b>13</b>

# 1 Introduction

## 1.1 What is Melt?

**Motivation**  $\LaTeX$  is great to format text. OCaml is great to program. What if you want to program  $\LaTeX$  documents using the OCaml syntax and type system? Then you can use Melt. Let's see how we can produce the following:

```
Let  $X \subset \{1, 2, 3\}$ .
```

With  $\LaTeX$ , this is easy:

```
Let  $X \subset \{1, 2, 3\}$ .
```

What does it mean to program this in OCaml? Let's assume that you have some `Latex` library providing the following functions: `t` which converts a string to  $\LaTeX$  text, `m` which converts a string to  $\LaTeX$  math, some basic  $\LaTeX$  macros and `c` which concatenates a list of  $\LaTeX$  expressions. Our example may then be encoded like this:

```
c [ t "Let "; m "X"; subset; lbrace; m "1, 2, 3"; rbrace; t "." ]
```

This is much more verbose than the original  $\LaTeX$  code! The Melt pre-processor allows lighter syntax:

```
"Let  $X \{subset\} \{1, 2, 3\}$ ."
```

Note how close this is to the  $\LaTeX$  syntax. The main differences are: usage of double quotes to enter *text* mode and usage of braces to enter *code* mode. The code mode, here, is used for command `subset`, which is an OCaml value of  $\LaTeX$  type.

**What is Melt?** Melt allows you to write  $\LaTeX$  documents using OCaml. It is composed of the `Latex` library for OCaml, the Melt preprocessor, the Melt tool, and the Melt library.

The `Latex` library provides an OCaml interface to  $\LaTeX$ . It allows to create  $\LaTeX$  values in text or math mode and to concatenate them. It defines basic  $\LaTeX$  commands as OCaml values, such as `subset`.

It is possible to write an entire document using only the `Latex` library, but the syntax of OCaml is awful for this purpose, and the document quickly becomes an awful, unreadable mess. The Melt preprocessor provides a much lighter syntax, allowing the user to easily interleave  $\LaTeX$  values and OCaml code.

The Melt library contains various tools to glue things together. In particular, it contains functions to easily include MLPOST pictures.

The Melt tool takes a Melt program and compiles it into a Postscript or a PDF document. It runs the preprocessor if needed, compiles the program, executes it to produce a  $\LaTeX$  program, and then runs  $\LaTeX$  to produce the final document.

## 1.2 Getting Started

A basic Melt program, `hello.mlt`, looks like this:

```
emit (document "Hello, world!")
```

Let's see how to compile this document, and how does it work.

**Compiling Using the Melt Tool.** You can compile `hello.mlt` in one command using the Melt tool:

```
melt hello.mlt
```

This produces a file `hello.ps` containing the text "Hello, world!". You can produce a PDF file instead, using the `-pdf` option.

**Compiling by Hand.** Instead of using the Melt tool, you can apply the preprocessor yourself:

```
meltpp hello.mlt -o hello.ml -open Latex -open Melt
```

This produces a file `hello.ml` which looks like this:

```
open Latex;;
open Melt;;
emit (document (mode T ((text "Hello, world!"))))
```

As you can see, the string `"Hello, world!"` is no longer a string but a value of type `Latex.t`. You can then compile this OCaml program. Once executed, it will produce a file `hello.tex` looking like this:

```
\documentclass{article}
\author{}
\begin{document}
  Hello, world!
\end{document}
```

You can compile this file using `latex` or `pdflatex`.

**Understanding the Code.** The `emit` function of the Melt library takes a value of type `Latex.t`. Values of this type describe L<sup>A</sup>T<sub>E</sub>X abstract syntax trees (ASTs), *i.e.*, pieces of L<sup>A</sup>T<sub>E</sub>X documents. The function writes this AST into the file `hello.tex`.

What we want to emit is a full L<sup>A</sup>T<sub>E</sub>X document. The `document` function of the `Latex` library takes a body, of type `Latex.t`, and returns another value of type `Latex.t`. The returned AST contains the document class, the `document` environment, the author and so on.

To produce the body of the document, we use the Melt preprocessor. In a Melt document, double quotes do not create values of type `string` but values of type `Latex.t`. So, here, `"Hello, world!"` is not a `string` but a piece of L<sup>A</sup>T<sub>E</sub>X AST that we can give to the `document` function.

**Text and Math Modes.** You can insert math formulas using the dollar delimiters `$\cdot\cdot\cdot`, as in L<sup>A</sup>T<sub>E</sub>X:

```
"Assume  $x+y=42$ ."
```

This will produce:

```
Assume  $x + y = 42$ .
```

The word “Assume”, as well as the ending dot, are printed in *text* mode, whereas the math formula is printed in *math* mode. You enter text mode using double quotes `"\cdot\cdot\cdot"`, and you enter math mode using dollars `\cdot\cdot\cdot`. Note that you can enter math mode directly:

```
emit (document  $x+y=42$ )
```

You can also enter text mode while in math mode:

```
 $x+y=42$  " but "  $z+t=69$ 
```

This will produce:

```
 $x + y = 42$  but  $z + t = 69$ 
```

These modes can be nested arbitrarily, but do not abuse this feature as your code will become less readable.

**Code Mode.** The third mode is called *code* mode. It is introduced using round brackets delimiters `{...}`. It allows to insert an OCaml value of type `Latex.t`. In particular, this is useful to write macros:

```
let p = $3.141592$ in
emit (document "{pi} is {p}.")
```

This will produce:

```
π is 3.141592.
```

`pi` is a value of type `Latex.t` which is defined in the `Latex` module.

In code mode, you can insert OCaml code of any complexity:

```
$1+2+3 = {latex_of_int (1+2+3)}$
```

This will produce:

```
1 + 2 + 3 = 6
```

The code in code mode is also preprocessed. This means that you can nest code, math and text modes arbitrarily. For example:

```
"This {emph "word"} is emphasized."
```

This will produce:

```
This word is emphasized.
```

**Verbatim Mode.** The last mode is called *verbatim* mode. It can be used to insert text that should not be parsed, such as code listing<sup>1</sup>. Here is an example:

```
"Here is some code: <<printf "Toto has $%d to spare." 42>>"
```

This will produce:

```
Here is some code: printf "Toto has $%d to spare." 42
```

As you can see, the code between the `<<...>>` delimiters is not preprocessed: the double quotes and dollars are left as it. Moreover, every symbol is translated into the latex command `symbol` so it can be printed correctly. Spaces and new lines are also translated. To sum up, the text you write in verbatim mode will be printed verbatim<sup>2</sup>.

Note that the `symbol` command of L<sup>A</sup>T<sub>E</sub>X often does not work well unless you use a `tt` font. The above example should be rewritten:

```
"Code: {texttt "<<printf "Toto has $%d to spare." 42>>}"
```

This will produce:

```
Code: printf "Toto has $%d to spare." 42
```

Also note that the verbatim mode can only be used in text mode. If you write `<<` in math mode, this will produce `<<`, and in code mode this will produce a syntax error from the OCaml compiler or a quotation if you use the `Camlp4` syntax extension.

The verbatim mode has other features such as delimiter selection, function selection and antiquotations. They will be described later in this document.

## 2 The Melt Distribution

The Melt distribution contains the `Latex` library, the Melt preprocessor, the `Melt` library, the Melt tool, and this documentation.

---

<sup>1</sup>It is *heavily* used in the source of this document, which is of course written using Melt.

<sup>2</sup>This is no coincidence.

## 2.1 Download

The Melt distribution can be found on OcamlForge:

```
http://melt.forge.ocamlcore.org/
```

Extract the archive wherever you want to obtain the `melt` directory.

## 2.2 Pre-requisites

You need the OCaml compiler. Version 3.09 is enough, maybe some previous versions will work too. Version 3.10.2 is needed if you want to compile using Ocamlbuild. Version 3.11 is needed if you want to be able to use native plugins for the Melt Preprocessor.

To compile Melt documents you need a L<sup>A</sup>T<sub>E</sub>X distribution.

To compile Melt documents which use Mlpost figures you need the Mlpost library. Versions from 0.6 to 0.7.4 are compatible. You will also need its dependencies, such as Metapost and the `context` package (Ubuntu, Debian, ...). Mlpost can be found at:

```
http://mlpost.lri.fr
```

You can compile and use Melt without Mlpost, but you will need to compile your Melt documents using the `-no-mlpost` option. If you later install Mlpost and want to use it with Melt you will have to recompile Melt.

## 2.3 Configure

The configuration tool is automatically launched the first time you run `make`. You can rerun it at any time by removing the `Config` file and running `make` again, or by running the following command in the `melt` directory:

```
ocaml configure.ml -i
```

The `-i` option activates interactive mode. If you don't use it, default values will be used.

You can also edit the `Config` file by hand.

## 2.4 Compile

Just enter the `melt` directory and run:

```
make
```

## 2.5 Install

Just enter the `melt` directory and run:

```
make install
```

You might have to add `sudo` at the beginning if do not have the right permissions on the installation directories.

## 2.6 Uninstall

Just enter the `melt` directory and run:

```
make uninstall
```

Be careful not to change the configuration in the `melt` directory between installing and uninstalling.

## 3 The Latex Library

The Latex library is composed of a kernel, which allows to describe low-level L<sup>A</sup>T<sub>E</sub>X ASTs, and of basic function definitions built on top of this kernel.

### 3.1 Using the Library

#### 3.1.1 Compiling the Library

The library is automatically compiled when you compile the Melt distribution (see Section 2).

#### 3.1.2 Installing the Library

The library will automatically be installed when you install the Melt distribution. It will normally be installed in the OCaml library directory, which you can find using the command `ocamlc -where`, under the `melt` subdirectory. The following files should be copied there:

```
latex.a
latex.cmi
latex.cma
latex.cmxa
```

To uninstall the library, simply delete these files or uninstall the Melt distribution (see Section 2).

#### 3.1.3 Compiling a Program Which Uses the Library

At compile-time, the OCaml compiler needs to know where to find the interface `latex.cmi`. If this file has been installed in the default OCaml library directory, OCaml will find it automatically. Else, you need to include the directory using the `-I` option. For example, to compile `hello.ml` if the library has been installed in the `melt` subdirectory of the default directory:

```
ocamlc -c -I +melt hello.ml
```

At link-time, the OCaml compiler also needs to know that the archive `latex.cma`, along with its dependency `str.cma`, have to be linked with the program. Add all these archives in the command line, in the right order. Replace `.cma` with `.cmxa` if you use the native compiler. For example, to produce the `hello` bytecode executable:

```
ocamlc -I +melt str.cma latex.cma hello.cmo -o hello
```

#### 3.1.4 Ocaml doc Documentation

The Ocaml doc documentation is automatically generated when you compile the Melt distribution (see Section 2). It contains the list and documentation of every functions and values of the library. Open its index file in your browser:

```
_build/latex/latex.docdir/index.html
```

## 3.2 The Kernel

```
type t (* the type of LaTeX abstract syntax trees *)
```

The Latex library allows you to build L<sup>A</sup>T<sub>E</sub>X Abstract Syntax Trees (ASTs) from a small set of basic constructions: modes, concatenation, commands and environments. These ASTs can then be pretty-printed as L<sup>A</sup>T<sub>E</sub>X document parts. This section details these basic constructions. Examples do not use the Melt preprocessor: strings are not escaped.

### 3.2.1 Raw Text

```
val text: string → t
```

The `text` function makes an AST from raw L<sup>A</sup>T<sub>E</sub>X code. Normally, you only use it without any special symbol such as `$`, `{`, `}` or `\`, unless you want to write something which is not supported by the library. Instead, use `text` to write text or math formulas:

```
text "Hello everyone."  
text "1 + 2 = 5"
```

AST parts produced by `text` are printed simply by printing the string given as argument.

### 3.2.2 Modes

```
type mode = M | T | A  
val mode: mode → t → t
```

The `mode` function takes an AST and sets its *mode*:

- M for math mode;
- T for text mode;
- A for any mode.

If the argument already had a mode, it is converted. Math mode is converted to text mode using dollars `$ $` while text mode is converted to math mode using `mbox`. If the old or the new mode is A, no conversion is made.

For instance, mode T (`text "Some text."`), when used in text mode, will be printed as `Some text.` (no conversion is needed) but when used in math mode, it will be printed as `\mbox{Some text.}`. Similarly, mode M (`text "1 + 2 = 5"`)<sup>3</sup>, when used in text mode, will be printed as `$1 + 2 = 5$` but when used in math mode, it will be printed as `1 + 2 = 5`.

Usually, the `text` and `mode` functions are used together. The `text` function may be used without `mode` if the raw L<sup>A</sup>T<sub>E</sub>X part does not need a mode (for instance, it is usable in any mode, or it is used in a context when the mode makes no sense).

### 3.2.3 Concatenation

```
val concat: t list → t
```

The `concat` function takes a list of ASTs and concatenate them in the given order. If a mode is given to the concatenation (using mode T (`concat [ ... ]`) for instance), components of the list with a different mode are converted.

### 3.2.4 Commands and Environments

```
val command:  
  ?packages: (string * string) list →  
  string →  
  ?opt: mode * t →  
  (mode * t) list →  
  mode →  
  t
```

<sup>3</sup>Please note that Melt does not ensure that the contents of your document makes sense.

Usage: `command ~packages name ~opt arguments mode`

The `command` function produces the application of a L<sup>A</sup>T<sub>E</sub>X *command*, i.e. a backslash `\`, followed by the name of the command, followed (if any) by the parameters of the command in braces `{ }`.

The `packages` optional parameter is used to specify which packages (and their options as the second element of the pair) must be added to the prelude of the document when the command is used. The `document` function of the library will scan all the AST to list these packages and print the prelude accordingly. If you do not use the `document` function to produce your document, the `packages` parameter will have no effect.

The `name` parameter is the name of the command, for instance `lambda` or `texttt`.

The `opt` optional parameter can be used to provide an optional argument to the command, which will be printed in brackets `[ ]` before the other arguments. The mode is the mode in which the argument will be printed.

The `arguments` parameter is the list of arguments which are printed in braces. Their respective mode is the mode in which they will be printed.

The `mode` parameter is the mode of the resulting command, as if you applied the `mode` function to the resulting AST (which is, thus, not needed).

```
val environment:  
  ?packages: (string * string) list →  
  string →  
  ?opt: mode * t →  
  ?args: (mode * t) list →  
  mode * t →  
  mode →  
  t
```

Usage: `environment ~packages name ~opt ~args main_arg mode`

The `environment` function is similar to the `command` function, except that it produces *environments* instead of commands. An environment begins with `\begin{...}` and finishes with `\end{...}`, and has a main argument `main_arg` which is printed between the two. It may still, however, have an optional parameter `~opt`, and additional arguments `~args` which are printed in braces after the `\begin` command.

In a typical Melt document, you will not use `command` and `environment` very often. Indeed, there is a chance that the command or the environment you need has already been encoded in the library as an OCaml value. Here are some examples which are taken from the implementation of the library:

```
let lambda = command "lambda" [] M  
let texttt x = command "texttt" [T, x] T  
let displaymath x = environment "displaymath" (M, x) T  
let includegraphics filename =  
  command ~packages: ["graphicx", ""] "includegraphics" [A, filename] T
```

### 3.2.5 Printing the AST

```
val to_buffer: ?mode: mode → Buffer.t → t → unit  
val to_channel: ?mode: mode → out_channel → t → unit  
val to_file: ?mode: mode → string → t → unit  
val to_string: ?mode: mode → t → string
```

These functions allow you to print a L<sup>A</sup>T<sub>E</sub>X AST (i.e. a value of type `Latex.t`) as L<sup>A</sup>T<sub>E</sub>X code. Function `to_buffer` prints into a buffer of the standard library, function `to_channel` prints into a channel of the standard library (module `Pervasives`), function `to_file` takes a file name as argument and prints into this file (which is truncated if it already exists), and function `to_string` returns a new string containing the L<sup>A</sup>T<sub>E</sub>X code.

You can use these functions to print any AST, whether it was created from the `document` function or not. If you print an AST which was created using `document`, it can be compiled with `latex` or `pdflatex`. Not using `document` can be interesting if you only want to produce a part of your document using Melt, and then include it in a L<sup>A</sup>T<sub>E</sub>X document using the `\input` command.



The Melt library also contains a function `emit` which should be used instead if you are using the Melt tool (see Section 6).

### 3.3 Pervasive Definitions

The library provides several  $\text{\LaTeX}$  commands and environment as OCaml values, along with other handy functions. This section does not detail every of them; view the Ocamldoc documentation for the full list (see Section 5.1.4).

#### 3.3.1 References and Labels

When you refer to other sections, figures, tables or items, you should use *labels*. If you write the index of the section directly (such as "`Section~7`"<sup>4</sup>) and then change the order of sections, you will have to change this "7" everywhere in your code, and you might miss some of them.

```
type label
val label: ?name: string → unit → label
val ref_: label → t
```

Function `label` creates a new label, and `ref_` makes reference to a label. However, this is not enough as the label must be placed at the position it references. In  $\text{\LaTeX}$ , labels can be placed anywhere, which often leads to errors. In Melt, labels are placed on meaningful constructions: sections (including chapters, sections, subsections and subsubsections) and figures (including other floats). These constructions are available as functions of the `Latex` library and they all have an optional argument `~label`.

Here is an example, written using the Melt preprocessor, of a section which refers itself:

```
let lbl_intro = label ()
let intro = section ~label: lbl_intro "This is Section~{ref_ lbl_intro}."
```

A good practice is to put all label declarations of your document (such as `lbl_intro`) in the same place, and use some consistent naming convention such as `lbl_something`.

You can still place labels anywhere using function `place_label`:

```
val place_label: label → t
```

However, this should only be used if you want to use some  $\text{\LaTeX}$  feature which is not implemented in Melt, or if you want to implement a command or environment similar to `section` or `figure`.

If you want to mix  $\text{\LaTeX}$  and Melt files by `\inputing .tex` files produced by Melt in your `.tex` documents, you will want to name your labels using the `~name` optional argument of the `label` function. This name corresponds to the argument of the `\label` command of  $\text{\LaTeX}$ . There are two possible reasons for this. Either the label has been placed in the  $\text{\LaTeX}$  code, and you want to refer to it in your Melt code; or the label has been placed in the Melt code, and you want to refer to it in your  $\text{\LaTeX}$  code.

If you do not wish to use the labels of Melt for some reason, it is easy to reimplement them in a way which is closer to  $\text{\LaTeX}$  labels:

```
let label x = command "label" [A, x] A
let ref_ x = command "ref" [A, x] A
```

If you use the preprocessor, don't forget to escape double quotes (`\`) as the command name is a string and not a `Latex.t`.

#### 3.3.2 Verbatim

The `Latex.Verbatim` module is very powerful, especially used in conjunction with the Melt preprocessor (see Section 5), as it allows:

- to write text that will be printed verbatim (i.e. "unchanged");

---

<sup>4</sup>Tilde `~` is a non-breakable space.

- to define pretty-printers of various languages;
- to define your own “mode”.

The last feature will be detailed in Section 5.

**Verbatim Functions** In Melt, a *verbatim function* is a function of type `string → Latex.t`, i.e. a function that somehow converts a string to a L<sup>A</sup>T<sub>E</sub>X AST. The `Latex.Verbatim` module provides several tools to build such function.

Function `Latex.Verbatim.verbatim` is a verbatim function which converts all non-alphanumerical characters using the `\symbol` command of L<sup>A</sup>T<sub>E</sub>X. This allows you to print *any* string without messing with L<sup>A</sup>T<sub>E</sub>X special characters such as `$`, `\`, `{` or `}`. Moreover, characters such as `+` have a special meaning in L<sup>A</sup>T<sub>E</sub>X. Spacing rules are applied to them. For instance, `$1+1$` is printed as `1 + 1`: space is added to make the formula more pretty. However, it is not convenient when you want to print a series of symbols, or a source code. The `verbatim` function solves this problem. Note, however, that the `\symbol` command of L<sup>A</sup>T<sub>E</sub>X does not work with fonts other than `tt`, so you should usually apply `texttt` to the result. For instance, `texttt (verbatim "$1+1$")` is printed as `$1+1$`.<sup>5</sup>

You can build other verbatim functions using `regexps`:

```
val regexps: (Str.regexp * (string → t)) list → (string → t) → string → t
```

Use the `regexps` function to apply a function to the parts of the string which match some regular expression. You can give several such functions along with their respective regular expressions, which are defined using the `Str` module of the standard library. The second argument is apply to parts which do not match any regular expression. This is the most general way of specifying verbatim functions that is provided by the library. If you need more, you can use other parser generators such as `Ocamllex` with `Ocamlyacc` or `Menhir`.

The second function is `keywords`:

```
val keywords: ?apply: (t → t) → string list → string → t
```

This is a particular instance of `regexps` which allows you to quickly build a verbatim function which simply applies `~apply` to all occurrences of a list of words. For instance, `keywords ["let"; "in"]` is a verbatim function which will apply `textbf` (bold font) to all occurrences of `let` and `in`.

The last function is `pseudocode`:

```
val pseudocode:
  ?trim: (string → string) →
  ?id_regexp: Str.regexp →
  ?kw_apply: (t → t) →
  ?id_apply: (t → t) →
  ?rem_apply: (string → t) →
  ?keywords: string list →
  ?symbols: (string * t) list →
  ?keyword_symbols: (string * t) list →
  ?underscore: Str.regexp →
  string → t
```

This function is a compromise between `regexps`, which can feel a bit heavy because of regular expressions, and `keywords`, which can only handle one kind of keywords as only one function can be applied. On the other hand, `pseudocode` can handle many kinds of replacement rules without requiring the use of regular expressions.

What `keywords` does is first separate *identifiers*, which are substrings matching `id_regexp`, from the rest of the string. Identifiers can then be *keywords* if they match a string of the `keywords` argument; in

<sup>5</sup>The `Latex.Verbatim.verbatim` function is more powerful than the L<sup>A</sup>T<sub>E</sub>X `\verb` command and `verbatim` environments. The first reason is that these do not work everywhere, whereas the `\symbol` function works as long as you can use a `tt` font. The second reason is that `\verb` and `verbatim` do not do exactly the same thing, which means that you cannot directly replace one with the other.

this case, `kw_apply` is applied to them. Identifiers can also be *keyword symbols*, which are keywords which will be replaced by a symbol. They are defined by the list of couples `keyword_symbols`. For instance, use the couple (`"nothing"`, `emptyset`) to replace each occurrences of “nothing” by  $\emptyset$ . Identifiers which are not keywords nor keyword symbols are just identifiers, and `id_apply` is applied to them.

Identifiers are split according to the `underscore` argument, which matches the underscore character by default. The first part is displayed normally, but the other parts are treated as indexes separated by commas. For instance, `hello_42_z` becomes *hello<sub>42,z</sub>*. Each part itself is checked to see if it is a keyword or a keyword symbol, which allows to replace, for instance, `hello_nothing` by *hello<sub>∅</sub>*.

Parts of the string which are not identifiers are then scanned for *symbols*, defined using the `symbols` argument in a similar fashion than keyword symbols. For instance, this can be used to replace `->`, which is not an identifier, to an arrow  $\rightarrow$ . Finally, everything which is not a keyword nor a symbol is replaced by applying `rem_apply`.

The string given to a pseudocode verbatim function is *trimmed* before being processed: empty lines at the beginning and at the end of the string are deleted. This allows you to start your code at the beginning of a new line of your source code, which is prettier. You can change this behavior by changing the `trim` argument, which will be applied instead of the default trimming function.

**Example: Boolean Formulas** Let’s use pseudocode to take boolean formulas such as:

$$A \wedge B \vee (C_1 \text{ xor } C_2) \Leftrightarrow (D \Rightarrow E_1 \text{ xand } E_2)$$

and print them like this:

$$A \wedge B \vee (C_1 \oplus C_2) \iff (D \Rightarrow E_1 \otimes E_2)$$

Operators `xor` and `xand` are identifiers which should be viewed as keywords printed as symbols, so we define them using the `~keyword_symbols` argument. Operators `/\`, `\/`, `<=>` and `==>` are not identifiers, they are simply symbols, so we define them using the `~symbols` argument. The result is the following function:

```

let boolean_formula =
  Verbatim.pseudocode
  ~symbols: [
    "/\\", land_;
    "\\\/", lor_;
    "<=>", iff;
    "==>", rightarrow_;
    "<==", leftarrow_;
  ]
  ~keyword_symbols: ["xor", oplus; "xand", otimes]

```

**Example: OCaml Pretty-Printer** Here is the verbatim function which is applied to all OCaml examples of this user manual:

```

let verbatim_keywords =
  Latex.Verbatim.keywords ~apply: (fun _ → texttt (symbolc ' _')) ["_"]

let ocaml_code_base x =
  Verbatim.pseudocode
  ~trim: (fun s → s)
  ~underscore: (Str.regexp "__")
  ~id_apply: (fun i → textsf (verbatim_keywords (to_string i)))
  ~kw_apply: (fun x → textbf (textsf x))
  ~rem_apply: (fun s → texttt (Latex.Verbatim.verbatim s))
  ~keywords: ["let"; "in"; "val"; "fun"; "type"]
  ~symbols: ["->", rightarrow]
  x

let ocaml_code x =
  Verbatim.regexp [
    Str.regexp "\\\"\\([\\|\\\"\\\\|\\^\\\"\\\\)*\\\"",
      (fun s → texttt (Latex.Verbatim.verbatim s));
    Str.regexp "(\\\"*\\([\\^*]\\\\|\\\\*[^\\\"\\\\])*\\\"*\\\"*\"",
      (fun s → textit (text s));
  ] ocaml_code_base (Verbatim.trim ['\n'] x)

```

Function `ocaml_code` first deals with string constants and comments using regular expressions. Note that this is not perfect, as it does not handle double quotes in comments. The rest of the code is processed by `ocaml_code_base`, which defines how keywords, symbols and identifiers should be printed. Several techniques are used.

- We set `~id_apply` and `~kw_apply` to print identifiers and keywords using a sans serif font (using `textsf`).
- OCaml identifiers tend to use underscores a lot, but `pseudocode` uses underscores to parse indices. We change this behavior by changing the `~underscore` optional parameter.
- Now that underscores are normal parts of identifiers, we need to print them correctly. So we change how keywords are printed, by using an auxiliary verbatim function `verbatim_keywords` which views underscores inside keywords, as keywords which should be printed as `tt` underscores (default underscore is `_`, and `tt` underscore is `_` which is prettier). Note that we could have simply used `Latex.Verbatim.verbatim` if identifiers were printed using a `tt` font instead of a sans serif font.
- Function `ocaml_code_base` will be applied to each part of the original string which is not a string constant. For instance, when we apply `ocaml_code` to `print "this";`, `ocaml_code_base` is actually applied twice: to `print "` and to `;`". This means that both these strings are trimmed, which is not the expected behavior. So we disable trimming by changing the `~trim` optional parameter, and we trim the string ourself in `ocaml_code` using `Verbatim.trim`.

## 4 The Melt Preprocessor

**TODO:** - modes - pragmas - verbatim (reire subsub-section verbatim) - command line options - plugins

## 5 The Melt Library

### 5.1 Using the Library

#### 5.1.1 Compiling the Library

The library is automatically compiled when you compile the Melt distribution (see Section 2).

### 5.1.2 Installing the Library

The library will automatically be installed when you install the Melt distribution. It will normally be installed in the OCaml library directory, which you can find using the command `ocamlc -where`, under the `melt` subdirectory. The following files should be copied there:

```
melt.a
melt.cmi
melt.cma
melt.cmxa
```

To uninstall the library, simply delete these files or uninstall the Melt distribution (see Section 2).

### 5.1.3 Compiling a Program Which Uses the Library

At compile-time, the OCaml compiler needs to know where to find the interface `melt.cmi`. If this file has been installed in the default OCaml library directory, OCaml will find it automatically. Else, you need to include the directory using the `-I` option. For example, to compile `hello.ml` if the library has been installed in the `melt` subdirectory of the default directory:

```
ocamlc -c -I +melt hello.ml
```

At link-time, the OCaml compiler also needs to know that the archive `melt.cma`, along with its dependencies `latex.cma`, `str.cma` and `mlpost.cma` (if you compiled Melt with MLPOST), have to be linked with the program. Add all these archives in the command line, in the right order. Replace `.cma` with `.cmxa` if you use the native compiler. For example, to produce the `hello` bytecode executable:

```
ocamlc -I +melt -I +mlpost mlpost.cma str.cma latex.cma melt.cma hello.cmo \
-o hello
```

Note that MLPOST itself may require some other dependencies.

### 5.1.4 Ocamldoc Documentation

The Ocamldoc documentation is automatically generated when you compile the Melt distribution (see Section 2). It contains the list and documentation of every functions and values of the library. Open its index file in your browser:

```
_build/melt/melt.docdir/index.html
```

## 5.2 .....

**TODO:** library documentation

## 6 The Melt Tool

**TODO:** - command line arguments - emit