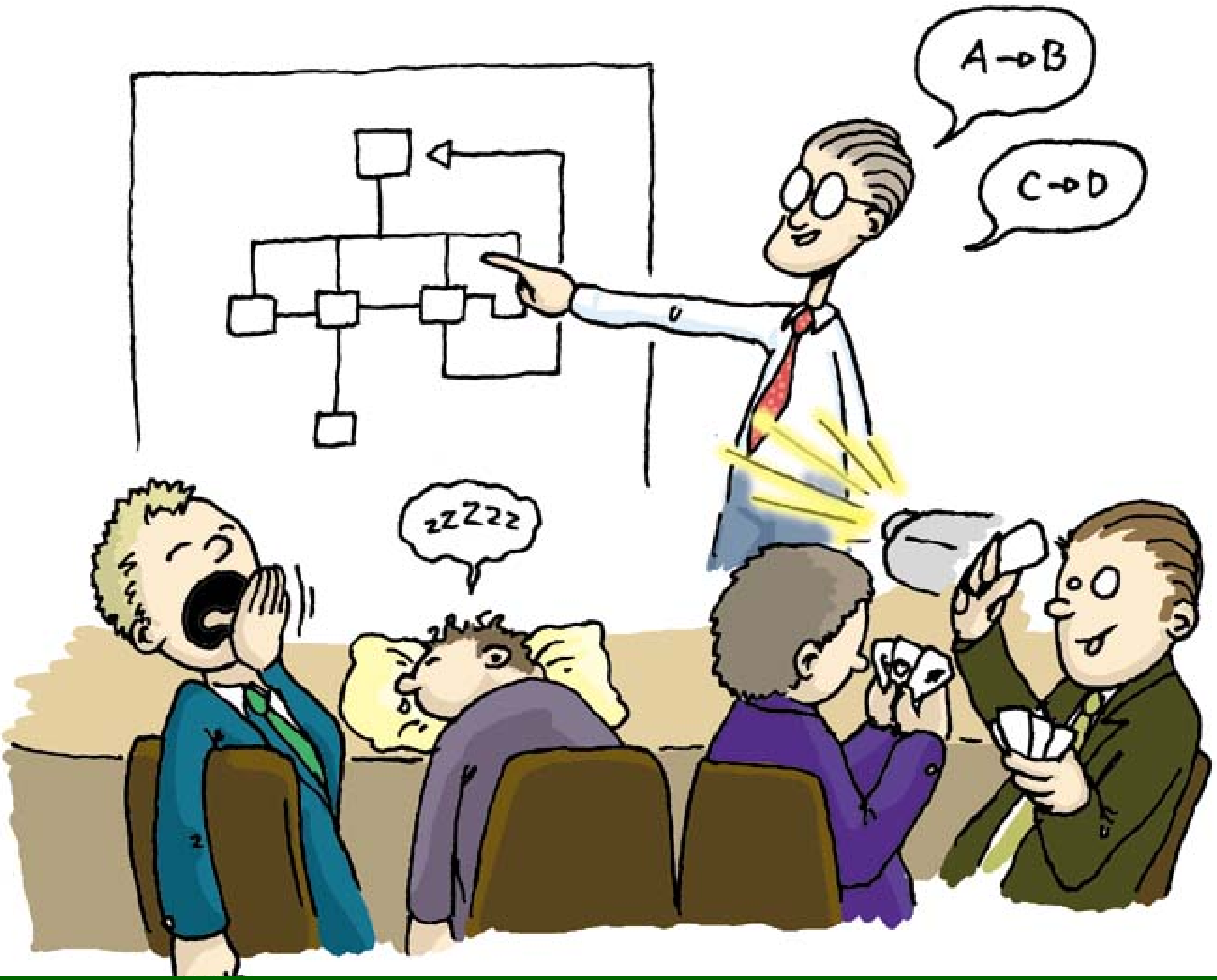
(How to prepare interesting and attractive teaching material)

How to use package MultEdu

János Végh



Abstract

For teaching my own courses, I developed a set of macros, since to display the course material under different circumstances different forms of teaching materials are needed. On the lectures I present the theoretical material in form of slides, and the explanation referring to the slides (of course in somewhat compressed form) I offer for my students, in a booklet-like form. My students studies that material either from printed hard copies, or from screen, using a browser or sometimes on mobile devices. The field is in continuous development, so I need to develop my teaching material also continuously. Because of this, it is a must to develop those different forms of the material synchronously. The simplest way to do so, is to use the same source, with proper formatting instructions. It is a serious challange, to develop course material for the today's students, who are used to lessons with high resolution, attractive graphics, good computer background.

As a common base, I used LaTeX, from which I produce the slides from the lectures using package beamer, and the reading material using package 'memoir'. This latter one can even reach the "on demand printing" quality. The printed material attempts to catch the attention with attractive graphical appearance, using above the average amount of figures (of course on can make also 'book-like' book, too). The booklet-like version contains all figures from the lectures, and some comprehensive version of the text of the lecture. The same text appears in screen-oriented form in the WEB-book format, and in the eBook compatible (native PDF) format.In those two forms (mainly targeting the small-screen mobile devices) bigger fonts are used and one figure/screen is displayed.

To satisfy those, somewhat contradictional requirements, one must make bargains,and more time and care must be invented in the formatting. The macros allow to support also foreign languages, and even to prepare course in English and your own language side by side. Using the possibilities of LaTeX, animations, movies, web-pages, sound files, etc. can also be embedded, but one has to think about the equivalent appearance on hard copies.

First edition: July 2016

V0.02 (using MultEdu V0.02) @2016.07.15
10 09 08 07 06 05 04 03 02 01 19 18 17 16 15 14 13

Contents

Chapter 1 Organizing the material

1.1 Document frame

1.1.1 Frontmatter

1.1.2 Mainmatter

1.1.3 Backmatter

1.2 Document sectioning

chapter, section,

Chapter 2

Sectioning document

2.1 Document units

Basically, the document is organized as 'beamer' needs it, but to print it in a book-like form, the sectioning must be changed, and also the package 'beamerarticle' must be used. In order to provide a uniform wrapper around sectioning, MultEdu introduces its own sectioning units.

2.1.1 Frames

These units actually correspond to the ones used in format 'book', and MultEdu transforms them properly when preparing slides.

Usage:

`\MEframe[keys]{subtitle}{content}`

Legal keys are

`shrink=true|false` and `plain=true|false`

By default, both are false.

2.1.2 Chapter

Correspondingly, the biggest unit is the 'chapter'. (As mentioned, for slides it is transformed to 'section'.) Usage:

`\MEchapter[short title]{long title}`

2.1.3 Section and below

The next, smaller unit is the 'section'. (As mentioned, for slides it is transformed to 'subsection'.) Usage:

`\MEsection[short title]{long title}`

In a similar way, there exists `\MEsubsection[short title]{long title}` and `\MEsubsubsection[short title]{long title}`; the latter one is transformed for slides to `\paragraph`

2.2 Dual language sources

It happens, that I teach the same course in my mother tongue for my domestic students, and in English, for foreign students. The course material is the same, and it must be developed in parallel. Obviously it is advantageous, if they are located in the same source file, side by side; so they can be developed in the same action. The `\UseSecondLanguage` macro supports this method.

The macros introduced above have a version with prefix 'MED' rather than 'ME only, which takes double argument sets (arguments for both languages). Depending on whether `\UseSecondLanguage` is defined, the first or the second argument set is used. In this way, just the 'src/Heading.tex' source file shall be edited, and the other language output will be prepared.

2.2.1 Switching between languages

Usage:

`\UseSecondLanguage{YES}`

where the argument `{}` is not relavant, only if this macro is defined or not. Traditionally, it is defined in 'src/Heading.tex'.

The two kinds of macros can be mixed, but only the 'D' macros are sensitive to changing the language.

2.2.2 Frames

In dual language documents, usually

`\MEDframe[keys]{subtitle, first language} {content, first language } {subtitle, second language} {content, second language}`

is used. I.e. the user provides titles and contents in both languages, and for preparing the output, selects one of them with `\UseSecondLanguage`.

2.2.3 Chapter

Correspondingly, the biggest unit in a dual language document is the 'Dchapter'. (As mentioned, for slides it is transformed to 'Dsection'.) Usage:

`\MEDchapter[short title1]{long title1}{short title2}{long title2}`

which is transformed to

`\MEchapter[short title1]{long title1}` or

`\MEchapter[short title2]{long title2}` calls,

depending on whether `\UseSecondLanguage` is or is not defined.

2.2.4 Section and below

The usage of the lower units is absolutely analogous.

2.3 Chapter illustration

Some book styles also allow presenting some illustration at the beginning of the chapters.

Usage:

```
\MEchapterillustration{file}
```

For slides, the illustration appears in a 'plain' style style. For books, the picture is placed at the beginning of the chapter. If the file name is empty, a 'fig/DefaultIllustration.png' file is searched. If the file not found, no illustration generated.

If macro `\DisableChapterIllustration` is defined, no picture generated.

Chapter 3

Preparing program listings

When teaching programming, it is a frequent need to display program listings. Through using package 'listings', MultEdu can implement this in very good quality. For details not described here see documentation of package 'listings'.

Notice that here the ratio of the listings within the text is unusually high, so it is very hard for the compiler to find good positioning. In the case of real texts, the page is much more aesthetic.

3.1 Setting appearance

Package 'listings' allows to set up the style of displaying program listings according to our taste (and the requirements). MultEdu pre-sets some style and allows to modify it as much as you like.

Macro

`\MESetStandardListingFormat` sets up a default appearance, and no programming language. Macro

`\MESetListingFormat[options]{language}`

sets the language, the same appearance as macro

`\MESetStandardListingFormat`

and also allows to overwrite parameters of 'listings' through 'options'.

3.2 Displaying inline fragments

A tipical task is to display a shorter fragment, like a line or a keyword. It is possible using `\lstinline|code|`.

The LaTeX commands appearing in this documentation are produced in such a way that at the beginning of the chapter commands

`\MESetListingFormat{TeX}`

```
\lstset{basicstyle=
\ttfamily\color{black}\normalsize}
```

or

```
\MESetListingFormat[basicstyle=
\ttfamily\color{black}\normalsize]{TeX}
```

are issued (otherwise the character size of the program text will be too small).

Listing 3.1: "Hello World" – a C++ way

```
#include <iostream>
using namespace std;
int
main ( int argc, char ** argv )
{
  // print welcome message
  cout << "Hello World" << endl;
  return 0;
}
```

3.3 Displaying program listings

Program listings can be displayed using macro

```
\MESourceFile[keys] {filename} {caption}
{label}{scale}.
```

The command used to display Listing 3.1 was

```
\MESourceFile[language={[ISO]C++}]
{lst/HelloWorld.cpp} {A "Hello World"- C++
program} {lst:hello.cpp}{}
```

Many times one needs wider program listings. In the case of the two-column printing, the listing shall fill the width of the two columns. The wide listings can be placed even hardly on the printed page (the first proper place, relative to the appearance of the macro is the top of the next page), and in addition, the orders of normal and wide listings cannot be changed. Because of this, the place where the listing appears, might be relatively far from the place of referencing it.

The command used to display Listing 3.2 :

```
\MESourceFile[language={[ISO]C++},wide]
{lst/HelloWorld.cpp} {A "Hello World"- C++
program, wide} {lst:Whello.cpp}{}
```

3.4 Decorations on listings

Different decorations can be placed on top of listings. To do so, one has to use the keyword `decorations`, and to insert as arguments the macros presented in this section.

The general form:

Listing 3.2: A "Hello World"- C++ program, wide

```
#include <iostream>
using namespace std;
int
main ( int argc, char ** argv )
{
  // print welcome message
  cout << "Hello World" << endl;
  return 0;
}
```

`\MESourceFile[options, decorations={ list of decorations } ] {source file} {caption} {label}{}`

where the list of decorations may contain any of the decoration macros presented in the section.

3.4.1 Highlighting lines

In a program listing, a range of source lines can be highlighted using macro

`\MESourcelinesHighlight{BallonName} {SourceName} {FirstLine} {LastLine}`.

Here `BallonName` is a new label, which denotes the newly created source line region highlighting box, `SourceName` is the label of the source file. That is, to highlight a program body in listing 3.3 the macro

```
\MESourceFile[language={[ISO]C++},
decorations={
\MESourcelinesHighlight{HelloBalloon}
{lst:HLhello.cpp} {6}{8} } ]
{lst/HelloWorld.cpp} {"Hello World" -- a C++
way, kijel~Olt} {lst:HLhello.cpp}{}
```

shall be used.

Listing 3.3: "Hello World" – a C++ way, highlighted

```
#include <iostream>
using namespace std;
int
main ( int argc, char ** argv )
{
  // print welcome message
  cout << "Hello World" << endl;
  return 0;
}
```

3.4.2 Commenting highlighted lines

The higlighting box can also be commented. Using macro

`\MESourceBalloonComment[keys]{BallonName} {ShiftPosition} {Comment} {CommentShape}`

allows to comment the balloon created previously. Here `BallonName` is the first argument of `\MESourcelinesHighlight`, `ShiftPosition` is the shift of the comment box, `Comment` is the comment text. Possible keys, with defaults are:

`width[=3cm]` and `color[=deeppeach]`.

Listing 3.4 is produced using macro

```
\MESourceFile[language={[ISO]C++},wide,
decorations={
\MESourcelinesHighlight{HelloBalloon}
{lst:HLChello.cpp} {6}{8}
\MESourceBalloonComment{HelloCBalloon}
{0cm,0cm} {This is the body} {CommentShape} }
] {lst/HelloWorld.cpp} {"Hello World" -- a
C++ way, commenting highlighted}
{lst:HLhello.cpp}{}
```

3.4.3 Commenting source lines

The individual source lines can also be commented, see Listing 3.5. To produce it, the command

```
\MESourceFile[language={[ISO]C++},wide,
decorations={
\MESourcelineComment{lst:Chello.cpp} {6}
{0cm,0cm} {This is a comment} {CommentShape}
} ]{lst/HelloWorld.cpp} {"Hello World" -- a
C++ way, commenting source lines}
{lst:Chello.cpp}{}
```

3.4.4 Numbered balls to listing

On the program listing numbered balls can also be located, for referencing the lines from the text. This can be done using macro

`\MESourcelineListBalls[keys]{ListingLabel}{List of lines}`

which puts a numbered ball at the end of the listed lines. Here `ListingLabel` is the label of the listing, `List of lines` is the list of sequence numbers of the lines to be marked. Possible key, with defaults:

`color[=orange]` and `number[=1]`.

Notes:

Listing 3.4: "Hello World" – a C++ way, commenting highlighted

```cpp
#include <iostream>
using namespace std;
int
main ( int argc, char ** argv )
{
  // print welcome message
  cout << "Hello World" << endl;
  return 0;
}
```

This is the body

Listing 3.5: "Hello World" – a C++ way, commenting source lines

```cpp
#include <iostream>
using namespace std;
int
main ( int argc, char ** argv )
{
  // print welcome message
  cout << "Hello World" << endl;
  return 0;
}
```

This is a comment

- When making slides, the balls will be put to separated slides.
- The positioning using geometrical positions, does not consider 'firstline'.

To produce Listing 3.6, the macro

`\MESourceFile[language={[ISO]C++}, decorations={ \MESourcelineListBalls{lst:LBhello.cpp}{3,8,5} } ] {lst/HelloWorld.cpp} {"Hello World" -- a C++ way, with balls} {lst:LBhello.cpp}{}`

has been used

Listing 3.6: "Hello World" – a C++ way, with balls

```cpp
#include <iostream>
using namespace std;
int ①
main ( int argc, char ** argv )
{ ③
  // print welcome message
  cout << "Hello World" << endl;
  return 0; ②
}
```

The marked lines can then be referenced through the balls like '(Listing 3.6 ②) is the return istruction'. It can be produced using

`\MEBall{Listing~\ref{lst:LBhello.cpp}}{2}`

3.4.5 Figure to listing

Sometimes one might need to insert figures into the listing. The macro is

`\MESourcelineFigure[keys] {SourceLabel} {LineNo} {ShiftPosition} {GraphicsFile}`. Possible key is `width[=3cm]`

To produce Listing 3.7, macro

`erilog\MESourceFile[language=V,wide, decorations={ \MESourcelineFigure[width=5.2cm] {lst:forloops.v}{8} {4.4,.0} {fig/forloops} } ] {lst/forloops.v} {Implementing \ctext{for} loop with repeating HW} {lst:forloops.v}{}`

was used.

3.5 Other related macros

3.5.1 Comparing source files

Sometimes it is worth to compare source files, side by side. The macro for this is

`\MESourceFileCompare[keys]{source file1} {source file2} {caption} {label}`

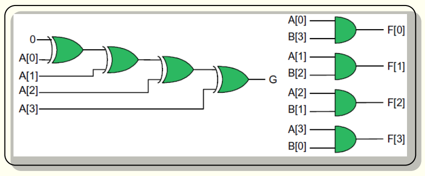
The command to produce Listing 22 is

`\MESourceFileCompare[language={[ANSI]C}] {lst/lower1.c} {lst/lower2.c} {Comparing two routines for converting string to lower case} {lst:lower12.c}`

Listing 3.7: Implementing `for` loop with repeating HW

```
// for == repeat HW

always @(A or B)
begin
  G = 0;
  for (I = 0; I < 4; I = I + 1)
  begin
    F[I] = A[I] & B[3-I];
    G = G ^ A[I];
  end
end
```



Listing 3.8: Comparing two routines for converting string to lower case

```
/* Convert string to lowercase: slow */
void lower1(char *s)
{
   int i;

   for (i = 0; i < strlen(s); i++)
   if (s[i] >= 'A' && s[i] <= 'Z')
    s[i] -= ('A' - 'a');
}
```

```
/* Convert string to lowercase: faster */
void lower2(char *s)
{
  int i;
  int len = strlen(s);
  for (i = 0; i < len; i++)

    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

It is also useful sometimes to show the source file with its output. The macro

```
\MESourceFileWithResult}[keys]{source file}
{result file} {caption} {label}
```

allows to do that. For producing Listing 3.10 the command

```
\MESourceFileWithResult[language=C++,wide,decorations={
\MESourcelineListBalls{lst:calculatorwithresult}
{13,14,16,18,19}
}] {lst/expensive_calculator.cpp}
{lst/calculatorresult.txt} {The
calculator program with its result}
{lst:calculatorwithresult}
```

was used.

3.6 Extra program languages

For my own goals, in addition to the programming languages defined in package 'listings', some further languages have been defined:

- diff
- [DIY]Assembler
- [ARM]Assembler
- [x64]Assembler
- [y86]Assembler

Listing 3.10: The calculator program with its result

```
// Expensive Calculator
// Demonstrates built-in arithmetic operators

#include <iostream>
using namespace std;

int main()
{
  cout << "7 + 3 = " << 7 + 3 << endl;
  cout << "7 - 3 = " << 7 - 3 << endl;
  cout << "7 * 3 = " << 7 * 3 << endl;

  cout << "7 / 3 = " << 7 / 3 << endl; 1
  cout << "7.0 / 3.0 = " << 7.0 / 3.0 << endl; 2

  cout << "7 % 3 = " << 7 % 3 << endl; 3

  cout << "7 + 3 * 5 = " << 7 + 3 * 5 << endl; 4
  cout << "(7 + 3) * 5 = " << (7 + 3) * 5 << endl; 5

  return 0;
}
```

```
7 + 3 = 10
7 - 3 = 4
7 * 3 = 21
7 / 3 = 2
7.0 / 3.0 = 2.33333
7 % 3 = 1
7 + 3 * 5 = 22
(7 + 3) * 5 = 50
```

Listings

Back cover