

MlDoc: automatic documentation extraction for Objective Caml

Pierre Boulet (Pierre.Boulet@lifl.fr)

Version 1.2.1

Contents

1 Usage: mldoc -help	1
2 Installation	1
2.1 Requirements	1
2.2 Compilation	1
3 Features	1
3.1 Code highlighting	1
3.2 Helping the programmer	2
3.3 Automatic title generation	2
3.4 Miscellaneous	3
4 Implementation	3
4.1 Code architecture	3
4.2 Constants	3
4.3 Code highlighting	4
4.4 Avoiding common L ^A T _E X errors in documentation comments	10
4.5 Main extraction function	11
4.6 Command line parsing	12
5 History	17
6 Acknowledgments	17

1 Usage: mldoc -help

2 Installation

2.1 Requirements

The following programs are required to compile MlDoc:

- Objective Caml version 3.08 (it may work with older versions)

- GNU Make
- L^AT_EX2e with packages alltt and color ; packages fontenc, inputenc and fullpage are needed to use option -main
- the documentation of mldoc uses the bera fonts ;
- H_EV_EA, version 1.04 (at least the developpement version from june 1999)

2.2 Compilation

First check the Makefile.config file for customization. Then, type make and make doc to generate this documentation in the doc directory.

To install, just copy the produced mldoc file where you want. The doc directory contains the postscript and html documentation.

3 Features

3.1 Code highlighting

- String are colored in grey (environment stre).
- Comments are italicized. MlDoc correctly handles nested comments (environment come).
- Keywords are typeset in boldface and blue (macro \kw).
- Value, type, module and class definitions are typeset in slanted (macro \definition).
- Preprocessing elements from Camlp4 are typeset in boldface and red (environment ppppe and macro \pppp):
 - quotations,
 - grammar extension keywords,
 - and the ifdef keyword.

Some other L^AT_EX macros are provided by option -main to be used in comments:

\code to typeset code,

\fun for the function names, and

\module for the module names.

3.2 Helping the programmer

A common error when writing documentation comments is forgetting to escape the “_” character. Thus MlDoc detects the incorrect uses of this character in the arguments of the following macros: \fun, \code, \module and \texttt. The following other characters are also automatically escaped (only if the \ has been forgotten): “^” and “#”.

3.3 Automatic title generation

Several options modify MIDoc's behavior with respect to title generation. The default option is `-auto`.

-title takes a string in argument to be used to format the title. Any `#l` character sequence will be replaced by the basename of the file in lower case, `#c` by the basename capitalized and `#u` by the basename in upper case, `#e` by the file extension and `##` by a single `#` character.

-notitle produce the same title as

```
-title '\label{#l#e}'
```

-intf produce the same title as

```
-title '\section{Module \module{#c} signature}
\label{#l.mli}'
```

-impl produce the same title as

```
-title '\section{Module \module{#c} implementation}
\label{#l.ml}'
```

-standalone produce the same title as

```
-title '\section{Standalone program \module{#c}}
\label{#l.ml}'
```

-ext if the file extension is `.ml`, equivalent to `-impl`, if it is `.mli`, equivalent to `-intf`, else produce the same title as

```
-title '\section{#l#e}
\label{#l#e}'
```

-auto as with option `-ext` but if both a signature file and an implementation file exist, process the two files (signature first).

3.4 Miscellaneous

- The `-main` option generates a compilable file with the default definitions for the needed macros and environments. It can be used as an example to customize the main file that will input the MIDoc generated files for each of the modules of a complex program.
- Everything before the first documentation comment is ignored. This allows to hide some unuseful comments and/or code in the documentation.
- To handle `ocamllex` and `ocamlyacc` files, comments enclosed between `/*TeX` and `*/` are recognized as documentation comments.
- Files ending with extension `.p4` are handled as implementation files. If both a `.p4` and a `.ml` file exist, the `-auto` file detection only considers the `.p4` one. This has been done to allow automatic preprocessing of `.p4` files by `camlp4`.

4 Implementation

4.1 Code architecture

mldoc is a standalone program that uses several modules from the standard library (Arg, Sys, Filename, String, Stack) and the Str library to handle regular expressions.

The algorithm is actually a finite state automaton (see section 4.5 below).

4.2 Constants

Values `verb` and `verbend` contain the text that delimitates the code in the output.

```
let verb =
  "\n\n%HEVEA\\begin{rawhtml}<blockquote><font size=-1>"
  ^"<hr width=25% height=2 noshade align=left>"
  ^"\\end{rawhtml}\\n"
  ^"%BEGIN LATEX\\n"
  ^"\\nopagebreak\\n"
  ^"\\begin{quotation}\\footnotesize\\noindent\\rule{5cm}{1pt}\\n"
  ^"%END LATEX\\n"
  ^"\\begin{alltt}\\n"

let verbend =
  "\\end{alltt}\\n"
  ^"%BEGIN LATEX\\n"
  ^"\\end{quotation}\\n"
  ^"%END LATEX\\n"
  ^"%HEVEA\\begin{rawhtml}</blockquote></font>\\end{rawhtml}\\n\\n"
```

Type `status` distinguishes the 3 states of the extraction function:

Skip initial state, everything is ignored till some documentation beginning,

TeX documentation,

CamL code.

The system state is memorized in reference state.

```
type status = Skip | TeX | CamL;;
let state = ref Skip;;
```

Type `context` indicates in which context the code analysis is done. Indeed, it is done with a stack automaton to be able to handle nested contexts such as nested comments or strings in comments... This stack is handled by functions `push`, `pop` and `get_context`.

```

type context = Code | Comment | String;;
let context : context Stack.t = Stack.create ();;
let push c = Stack.push c context;;
let pop () = let _ = Stack.pop context in ();;
let get_context () =
  let c = Stack.pop context in
    Stack.push c context;
  c;;

```

4.3 Code highlighting

Function `clean` highlights the definitions and escapes the tabulations and characters `\`, `{` and `}` which keep their meaning in the `alltt` environment. All this is done using regular expressions (from module `Str`).

```

open Str;;
let clean ligne =
  let rec tabify s =
    try
      let index_of_tab = String.index s '\t' in
      let rindex_of_tab = (String.length s) - index_of_tab - 1 in
      let nb_spaces = 8 - (index_of_tab mod 8) in
      let prefix = String.sub s 0 index_of_tab in
      let spaces = String.make nb_spaces ' ' in
      let suffix = if rindex_of_tab = 0 then
        "" else String.sub s (index_of_tab + 1) rindex_of_tab in
      tabify (prefix ^ spaces ^ suffix)
    with Not_found -> s
  and escape_braces s =
    global_replace (regexp "[{}&]") "\\\\\\\\0{}" s
  and escape_bs s =
    global_replace (regexp "\\\\") "\\verb!\\!" s
  in let clean s = escape_braces (escape_bs (tabify ligne))
  in
    let defs s =
      global_replace
        (regexp "\\b\\(let\\|type\\|and\\|class\\|module\\)\\b[ \\t][^=]*=")
        "\\definition{\\0}" s
    and vals s =
      global_replace
        (regexp "\\b\\(val\\|class\\|module\\)\\b[ \\t][^:=]*=")
        "\\definition{\\0}" s
    in let fonts s = vals (defs s)
  in fonts (clean ligne);;

```

As said before, highlighting is done with a stack automaton. Here is its implementation. The entry point of the automaton is function `convert`. All the states are coded by a function that converts a `char Stream.t` into a `String`.

One has to be careful of the treatment of the previously escaped characters `\`, `&`, `{` and `}` and of the definitions.

```

let soc c = String.make 1 c;;
let sep c =
  c=' ' or c=';' or c='.' or c=',' or c='[' or c=']'
or c='{ ' or c='| ' or c='=' or c='<' or c='>' or c=')'
or c=':' or c='#' or c='-' or c='?';;

let rec convert str =
  match get_context () with
  | Code -> convert_code str
  | Comment -> convert_comment str
  | String -> convert_string str
and convert_code =
  parser
  [< '""'; s >] ->
    push String;
    "\\begin{stre}\"""^(convert_string s)
  | [< ''(''; s >] -> paren_code s
  | [< ''\''; s >] -> quote1_code s
  | [< '<'; s >] -> lt_code "<" s
  | [< 'c; s >] ->
    if (('a'<=c && c<='z') or ('A'<=c && c<='Z') or
      ('À'<=c && c<='ÿ') or c='_')
    then ident (soc c) s
    else (soc c)^(convert_code s)
  | [< >] -> ""

```

Quotation treatment.

```

and lt_code buff =
  parser
  [< '<'; s >] -> quotation (buff^"<") s
  | [< '''; s >] -> lt1_code (buff^":") s
  | [< '""'; s >] ->
    push String;
    buff^"\\begin{stre}\"""^(convert_string s)
  | [< ''(''; s >] -> buff^(paren_code s)
  | [< ''\''; s >] -> buff^(quote1_code s)
  | [< 'c; s >] ->
    if (('a'<=c && c<='z') or ('A'<=c && c<='Z') or
      ('À'<=c && c<='ÿ') or c='_')
    then buff^(ident (soc c) s)
    else buff^(soc c)^(convert_code s)
  | [< >] -> buff
and lt1_code buff =
  parser
  [< '<'; s >] -> quotation (buff^"<") s
  | [< 'c; s >] ->
    if (('a'<=c && c<='z') or ('A'<=c && c<='Z') or
      ('À'<=c && c<='ÿ') or c='_')
    then lt1_code (buff^(soc c)) s
    else failwith "bad quotation ??"
  | [< >] -> buff
and quotation buff =
  parser
  [< '>'; s >] -> end_quotation (buff^">") s

```

```

| [< 'c; s >] -> quotation (buff^(soc c)) s
| [< >] -> buff
and end_quotation buff =
  parser
    [< ''>; s >] ->
      "\\begin{ppppe}"^buff^">\\end{ppppe}"^(convert_code s)
    | [< 'c ;s >] -> quotation (buff^(soc c)) s

```

Character treatment.

```

and quote1_code =
  parser
    [< ''\\'; s >] -> quote_bs_code s
    | [< ''\\''; s >] -> ""^(quote1_code s)
    | [< 'c; s >] -> quote2_code ("'"^(soc c)) s
    | [< >] -> failwith "quote at end_of_line ??"
and quote_bs_code =
  parser
    [< ''v'; 'e'; 'r'; 'b'; '!'; '\\';
      '!'; s >] ->
      quote_bsverb_code s
    | [< ''{'; ''{'; ''}'; '\\''; s >] ->
      "\\begin{stre}'\\{'}\\end{stre}"^(convert_code s)
    | [< ''}'; ''{'; ''}'; '\\''; s >] ->
      "\\begin{stre}'\\}'\\end{stre}"^(convert_code s)
    | [< ''&; ''{'; ''}'; '\\''; s >] ->
      "\\begin{stre}'\\&{}'\\end{stre}"^(convert_code s)
    | [< >] -> failwith "quote_bs_code"
and quote_bsverb_code =
  parser
    [< ''\\'; 'v'; 'e'; 'r'; 'b'; '!'; '\\';
      '!'; '\\''; s >] ->
      "\\begin{stre}'\\verb!\\!\\verb!\\!'\\end{stre}"^(convert_code s)
    | [< ''n'; '\\''; s >] ->
      "\\begin{stre}'\\verb!\\!n'\\end{stre}"^(convert_code s)
    | [< ''r'; '\\''; s >] ->
      "\\begin{stre}'\\verb!\\!r'\\end{stre}"^(convert_code s)
    | [< ''t'; '\\''; s >] ->
      "\\begin{stre}'\\verb!\\!t'\\end{stre}"^(convert_code s)
    | [< ''b'; '\\''; s >] ->
      "\\begin{stre}'\\verb!\\!b'\\end{stre}"^(convert_code s)
    | [< ''\\''; '\\''; s >] ->
      "\\begin{stre}'\\verb!\\!'\\end{stre}"^(convert_code s)
    | [< 'c0; 'c1; 'c2; '\\''; s >] ->
      if '0' <= c0 && c0 <= '9' &&
        '0' <= c1 && c1 <= '9' &&
        '0' <= c2 && c2 <= '9'
      then
        "\\begin{stre}'\\verb!\\!"^(soc c0)
        ^"(soc c1)^(soc c2)"^"\\end{stre}"^(convert_code s)
      else failwith "quote_bs_code"
    | [< 'c; '\\''; s >] ->
      "\\begin{stre}'\\verb!\\!"^(soc c)^"\\end{stre}"^(convert_code s)
    | [< >] -> failwith "quote_bsverb_code"
and quote2_code str =
  parser

```

```

    [< '\'; s >] ->
        "\\begin{stre}^str^"\\end{stre}^(convert_code s)
| [< '""; s >] ->
    push String;
    str^"\\begin{stre}\"\"^(convert_string s)
| [< '('; s >] -> str^(paren_code s)
| [< 'c; s >] -> str^(soc c)^(convert_code s)
| [< >] -> str

```

Identifier and keyword treatment.

```

and ident str =
  parser
    [< '""; s >] ->
      push String;
      (convert_ident str)^"\\begin{stre}\"\"^(convert_string s)
| [< '('; s >] -> (convert_ident str)^(paren_code s)
| [< 'c; s >] ->
    if (('0'<=c && c<='9') or c=='\'' or
        ('a'<=c && c<='z') or ('A'<=c && c<='Z') or
        ('À'<=c && c<='ÿ') or c=='_')
    then ident (str^(soc c)) s
    else if sep c
    then (convert_ident str)^(soc c)^(convert_code s)
    else str^(soc c)^(convert_code s)
| [< >] -> (convert_ident str)
and convert_ident str =
  if List.mem str ["and";"as";"asr";"begin";
                  "class";"closed";"constraint";
                  "do";"done";"downto";
                  "else";"end";"external";
                  "false";"for";"fun";"function";"functor";
                  "if";"in";"include";"inherit";
                  "land";"lazy";"let";"lor";"lsl";"lsr";"lxor";
                  "match";"method";"mod";"module";"mutable";"new";
                  "object";"of";"open";"or";"parser";"private";"rec";
                  "sig";"struct";"then";"to";"true";"try";"type";
                  "val";"virtual";"when";"while";"with"]
  then "\\kw{^str^}"
  else if List.mem str ["assert"; "exception"; "failwith"; "raise";
                        "invalid_arg"]
  then "\\exception{^str^}"
  else if List.mem str ["EXTEND"; "END"; "GLOBAL"; "FIRST"; "LAST";
                        "BEFORE"; "AFTER"; "LEVEL"; "SELF"; "NEXT";
                        "LIST0"; "LIST1"; "OPT"; "DELETE_RULE";
                        "LEFTA"; "RIGHTA"; "NONA"; "ifdef"]
  then "\\pppp{^str^}"
  else str

```

String treatment.

```

and convert_string =
  parser
    [< '""; s >] ->

```

```

    pop ();
    (match get_context () with
    | Code -> "\\end{stre}"^(convert_code s)
    | Comment -> ""^(convert_comment s)
    | String -> failwith "context: string in string ??")
| [< '\\'; s >] -> bs_string s
| [< 'c; s >] -> (soc c)^(convert_string s)
| [< >] -> ""
and bs_string =
  parser
    [< 'v'; 'e'; 'r'; 'b'; '!'; '\\';
    '!'; s >] -> escape_string s
  | [< '{'; '{'; '}''; s >] ->
    "\\{"^(convert_string s)
  | [< '}''; '{'; '}''; s >] ->
    "\\}"^(convert_string s)
  | [< '&'; '{'; '}''; s >] ->
    "\\&{"^(convert_string s)
  | [< 'd'; 'e'; 'f'; 'i'; 'n'; 'i'; 't'; 'i'; 'o'; 'n';
    '}''; s >] -> definition_in_string s
and escape_string =
  parser
    [< '\\'; 'v'; 'e'; 'r'; 'b'; '!'; '\\';
    '!'; s >] -> "\\verb!\\!\\verb!\\!"^(convert_string s)
  | [< 'c; s >] -> "\\verb!\\!"^(soc c)^(convert_string s)
  | [< >] -> ""
and escape_string_in_def =
  parser
    [< '\\'; 'v'; 'e'; 'r'; 'b'; '!'; '\\';
    '!'; s >] -> "\\verb!\\!\\verb!\\!"^(definition_in_string s)
  | [< 'c; s >] -> "\\verb!\\!"^(soc c)^(definition_in_string s)
  | [< >] -> ""
and definition_in_string =
  parser
    [< '}''; s >] -> convert_string s
  | [< '\\'; s >] -> bs_def_string s
  | [< 'c; s >] -> (soc c)^(definition_in_string s)
and bs_def_string =
  parser
    [< 'v'; 'e'; 'r'; 'b'; '!'; '\\';
    '!'; s >] -> escape_string_in_def s
  | [< '{'; '{'; '}''; s >] ->
    "\\{"^(definition_in_string s)
  | [< '}''; '{'; '}''; s >] ->
    "\\}"^(definition_in_string s)
  | [< '&'; '{'; '}''; s >] ->
    "\\&{"^(definition_in_string s)

```

Comment treatment.

```

and paren_code =
  parser
    [< '*'; s >] ->
      push Comment;
      "\\begin{come}*"^(convert_comment s)
  | [< '""'; s >] ->

```

```

    push String;
    "\\begin{stre}\"^(convert_string s)
| [< ''('; s >] -> "\"^(paren_code s)
| [< ''\''; s >] -> "\"^(quote1_code s)
| [< '<'; s >] -> "\"^(lt_code "<" s)
| [< 'c; s >] ->
    if (('a'<=c && c<='z') or ('A'<=c && c<='Z') or
        ('À'<=c && c<='ÿ') or c='_')
    then "\"^(ident (soc c) s)
    else "\"^(soc c)^(convert_code s)
| [< >] -> "("
and convert_comment =
parser
| [< '*'; s >] -> star_comment s
| [< ''''; s >] ->
    push String;
    "\"^(convert_string s)
| [< ''('; s >] -> paren_comment s
| [< 'c; s >] -> (soc c)^(convert_comment s)
| [< >] -> ""
and star_comment =
parser
| [< '''); s >] ->
    pop ();
    (match get_context () with
    | Code -> "*"\\end{come}\"^(convert_code s)
    | Comment -> "*"^(convert_comment s)
    | String -> failwith "context: comment in string ??")
| [< '*'; s >] -> "*"^(star_comment s)
| [< ''''; s >] ->
    push String;
    "*"^(convert_string s)
| [< ''('; s >] -> "*"^(paren_comment s)
| [< 'c; s >] -> "*"^(soc c)^(convert_comment s)
| [< >] -> "*"
and paren_comment =
parser
| [< '*'; s >] ->
    push Comment;
    "*"^(convert_comment s)
| [< ''''; s >] ->
    push String;
    "\\\"^(convert_string s)
| [< ''('; s >] -> "\"^(paren_comment s)
| [< 'c; s >] -> "\"^(soc c)^(convert_comment s)
| [< >] -> ""
;;

```

4.4 Avoiding common L^AT_EX errors in documentation comments

Dealing with forgotten `\` in function names. We detect the following macros : `\fun`, `\code`, `\module` and `\texttt`. Inside the arguments of these macros, some special characters are escaped : `_`, `^`, and `#`.

The global variable `in_macro` is used to deal with multiline macro arguments.

```

let in_macro = ref (-1);;

let rec escape =
  parser
    [< '\\\'; s >] -> is_macro "\\\" s
  | [< 'c; s >] -> (soc c)^(escape s)
  | [< >] -> ""
and is_macro buff =
  parser
    [< '{'; s >] ->
      if buff=="\fun" or buff=="\code" or buff=="\module"
      or buff = "\\texttt"

      then begin
        in_macro := 0;
        (buff^{^(macro s)})
      end else (buff^{^(escape s)})
  | [< '\\\'; s >] -> (buff^(is_macro "\\\" s))
  | [< 'c; s >] -> is_macro (buff^(soc c)) s
  | [< >] -> buff
and macro =
  parser
    [< '''; s >] ->
      if !in_macro=0 then begin
        in_macro := -1;
        "}^{^(escape s)}"
      end else begin
        decr in_macro;
        "}^{^(macro s)}"
      end
  | [< '{'; s >] -> incr in_macro; "{^{^(macro s)}"
  | [< '_'; s >] -> "\\_\"^{^(macro s)}"
  | [< '^'; s >] -> "\\^\"^{^(macro s)}"
  | [< '#'; s >] -> "\\#\"^{^(macro s)}"
  | [< '\\\'; s >] -> bs_macro s
  | [< 'c; s >] -> (soc c)^(macro s)
  | [< >] -> ""
and bs_macro =
  parser
    [< '_'; s >] -> "\\_\"^{^(macro s)}"
  | [< '^'; s >] -> "\\^\"^{^(macro s)}"
  | [< '#'; s >] -> "\\#\"^{^(macro s)}"
  | [< 'c; s >] -> "\\\"^{^(soc c)^(macro s)}"
  | [< >] -> ""
;;

```

4.5 Main extraction function

Function *extract* works as a finite state automaton. It has 3 states represented by functions *skipper*, *teckel* and *came*. *extract* iterates a line by line analysis until exception `End_of_file` is thrown. At this moment, an ending sequence is executed.

```

let extract in_chan out_chan =
  let remove_eol_spaces s =
    let last = ref (String.length s) in

```

```

    while !last > 0 && (s.[!last - 1] = ' ' or s.[!last - 1] = '\t') do
      decr last
    done;
    String.sub s 0 !last
  in
  let remove_bol_spaces s =
    let last = ref (String.length s) in
    let cur = ref 0 in
      while (!cur < !last) && (s.[!cur] = ' ' or s.[!cur] = '\t') do
        incr cur
      done;
      String.sub s !cur (!last - !cur)
    in
  let skipper lll =
    let lll_but_spaces = remove_bol_spaces lll in
    if (lll_but_spaces = "(*TeX)" || (lll_but_spaces = "/*TeX"))
    then begin
      in_macro := -1;
      TeX
    end else Skip
  and teckel lll =
    let lll_but_spaces = remove_bol_spaces lll in
    if (lll_but_spaces <> "*)" && (lll_but_spaces <> "*/")
    then begin
      if !in_macro = -1 then
        output_string out_chan (escape (Stream.of_string lll))
      else output_string out_chan (macro (Stream.of_string lll));
      output_char out_chan '\n';
      flush out_chan;
      TeX
    end else begin
      output_string out_chan verb;
      flush out_chan;
      Stack.clear context;
      push Code;
      Caml
    end
  and came lll =
    let lll_but_spaces = remove_bol_spaces lll in
    if (get_context () <> Code)
    or ((lll_but_spaces <> "(*TeX)" && (lll_but_spaces <> "/*TeX")) then
      begin
        output_string
          out_chan
            (convert (Stream.of_string (clean lll)));
        output_char out_chan '\n';
        flush out_chan;
        Caml
      end
    else begin
      output_string out_chan verbend;
      flush out_chan;
      in_macro := -1;
      TeX
    end
  in try
    while true do
      let ligne = remove_eol_spaces (input_line in_chan)

```

```

    in match !state with
      | Skip -> state := skipper ligne
      | TeX -> state := teckel ligne
      | Caml -> state := came ligne
    done
with End_of_file ->
begin
  close_in in_chan;
  match !state with
    | Skip -> ()
    | TeX -> ()
    | Caml ->
      ( Stack.clear context;
        output_string out_chan "  \n";
        output_string out_chan verband)
  end;;

```

4.6 Command line parsing

The main function, *main*, analyzes the command line (using module *Arg*) and calls *extract*. Function *maketitle* deals with the title generation.

```

type mode = Ext of string | Auto | Impl | Intf | Standalone
           | NoTitle | Title of string;;

let rec maketitle out_chan base file mode =
  let in_chan = open_in file in
  match mode with
  | Ext ".ml" ->
    if Sys.file_exists (String.sub file 0
                                ((String.length file) - 1))
    then maketitle out_chan base file Impl
    else maketitle out_chan base file Standalone
  | Ext ".p4" ->
    if Sys.file_exists (String.sub file 0
                                ((String.length file) - 1))
    then maketitle out_chan base file Impl
    else maketitle out_chan base file Standalone
  | Ext ".mli" -> maketitle out_chan base file Intf
  | Ext e ->
    maketitle out_chan base file
      (Title "\\section{#l#e\\label{#l#e}}")
  | Impl ->
    maketitle out_chan base file
      (Title "\\section{Module \\module{#c} implementation
              \\label{#l.ml}}\n")
  | Intf ->
    maketitle out_chan base file
      (Title "\\section{Module \\module{#c} signature
              \\label{#l.mli}}\n")
  | Standalone ->
    maketitle out_chan base file
      (Title "\\section{Standalone program \\module{#l}
              \\label{#l.ml}}\n")

```

```

| NoTitle ->
  maketitle out_chan base file (Title "\\label{#l#e}\n")
| Title t ->
  let s = String.rindex file '.' in
  let ext = String.sub file s ((String.length file) - s)
  in output_string out_chan
    (escape
      (Stream.of_string
        (global_replace (regexp "##") "#"
          (global_replace (regexp "#e") ext
            (global_replace (regexp "#l") base
              (global_replace
                (regexp "#c") (String.capitalize base)
                (global_replace
                  (regexp "#u") (String.uppercase base)
                  t)))))))));
    extract in_chan out_chan;
    print_string ("\tfrom "^file^\n")
| Auto -> invalid_arg "maketitle: Auto";

let main () =
  let file = ref ""
  and outfile = ref ""
  and mode = ref Auto
  and is_main = ref false in
  let speclist = [
    "-ext", Arg.Unit (fun _ -> mode := Ext ""),
    ": file type is inferred by its extension";
    "-auto", Arg.Unit (fun _ -> mode := Auto),
    ": (default) mldoc tests the existence of files with the same
    basename and treats all these files according their extension";
    "-impl", Arg.Unit (fun _ -> mode := Impl),
    ": file is interpreted as an implementation file";
    "-intf", Arg.Unit (fun _ -> mode := Intf),
    ": file is interpreted as an interface file";
    "-standalone", Arg.Unit (fun _ -> mode := Standalone),
    ": file is interpreted as a standalone program";
    "-notitle", Arg.Unit (fun _ -> mode := NoTitle),
    ": truns off the automatic title generation";
    "-title", Arg.String (fun t -> mode := Title t),
    "string : sets the title to string. Any #l character sequence
    will be replaced by the basename of the file in lower case,
    #c by the basename capitalized and #u by the basename in upper
    case, #e by the file extension and ## by a single # character.";
    "-o", Arg.String (fun f -> outfile := f),
    "filename : sets the output file name to filename";
    "-main", Arg.Unit (fun _ -> is_main := true),
    ": builds a compilable (standalone) LaTeX file"]
  and anonfun = fun s ->
    if !file = "" then file := s
    else raise (Arg.Bad "Too many arguments")
  and usage_msg =
    "Usage:\n"
    ^"mldoc [-main|-o file|-ext|-auto|-impl|-intf|-standalone
    | -notitle|-title t] [file]\n\n"
    ^"mldoc reads one Caml file (.ml, .mli, .mly, .mll) and extracts\n
    the LaTeX documentation it contains. This LaTeX file can\n
    later be treated by HeVeA to get an HTML documentation. mldoc\n

```

identifies the two character strings: `\>(*TeX\`, beginning of documentation, and `\(*)\`, end of documentation.
 All the text between these two marks is output without modification. These marks must be alone on a line.
 \n
 The text outside these marks is considered as code. \n
 It is composed in small typewriter font and is preceded by a rule to get the attention of the reader. Furthermore, the code is highlighted. Currently, strings, (nested) comments, keywords, preprocessing elements for `camlp4` and definitions are highlighted.
 \n
 If no file is present, the input will be the standard input and the output, the standard output. In this case, unless the `-title` option is present, no title will be generated. The default output filename is the basename of the input file with the `\.tex\` extension. It can be overridden with the `-o` option.
 \n
 See <http://www.lifl.fr/~{}boulet/softs.html> for more information.
 \n"

in

```
Arg.parse speclist anonfun usage_msg;
let print_header out_chan =
  if !is_main then output_string out_chan
    ("\\documentclass[a4paper,11pt]{article}
\\usepackage[T1]{fontenc}
\\usepackage[latin1]{inputenc}
\\usepackage{alltt}
\\usepackage{fullpage}
\\usepackage[dvips]{color}
"^(soc '\032')^"
%HEVEA\\makeatletter
%HEVEA\\renewcommand{\\@bodyargs}{BGCOLOR=white TEXT=black}
%HEVEA\\makeatother

%commands to use in comments to present the code
\\newcommand{\\module}[1]{\\textsf{#1}}
\\newcommand{\\code}[1]{\\texttt{#1}}
\\definecolor{kwc}{rgb}{0,0,0.8}
\\newcommand{\\kw}[1]{\\textcolor{kwc}{\\textbf{#1}}}
\\definecolor{exceptionc}{rgb}{1,0,0}
\\newcommand{\\exception}[1]{\\textcolor{exceptionc}{\\textbf{#1}}}
\\definecolor{ppppc}{rgb}{0.8008, 0.3594, 0.3594}
\\newcommand{\\pppp}[1]{\\textcolor{ppppc}{\\textbf{#1}}}
\\newenvironment{ppppe}{\\color{ppppc}\\bfseries}{}
\\newcommand{\\definition}[1]{\\texttt{\\textsl{#1}}}
\\newcommand{\\fun}[1]{\\definition{#1}}
\\definecolor{strc}{rgb}{0.5,0.5,0.5}
\\newenvironment{stre}{\\color{strc}}{}
\\newenvironment{come}{\\itshape}{}

\\title{Documentation for "
  ^(escape (Stream.of_string("\\texttt{^(!file)^"})) )^"}
\\date{\\today}
\\author{extracted by \\code{mldoc}}

\\begin{document}
\\maketitle{}
```

```

\n")
and print_trailer out_chan =
  if !is_main then output_string out_chan "\\end{document}\n";
  close_out out_chan
in
  if !file = "" then begin
    let outchan =
      if !outfile = "" then stdout
      else open_out !outfile in
      print_header outchan;
      (match !mode with
       | Title t -> print_endline t;
       | _ -> ());
      extract stdin outchan;
      print_trailer outchan;
      if !outfile != "" then close_out outchan
    end else
      let indirname = Filename.dirname !file
      and basename = Filename.basename !file in
      let base = Filename.chop_extension basename in
      let out_file =
        if !outfile = "" then base^".tex"
        else !outfile in
      print_string ("Building file "^out_file);
      print_char '\n';
      let out_chan = open_out out_file in
      begin
        print_header out_chan;
        state := Skip;
        (* Title generation mode selection. *)
        begin match !mode with
          | Ext "" ->
              let s = String.rindex basename '.' in
                maketitle out_chan base !file
                  (Ext (String.sub basename
                    s ((String.length basename) - s)))
          | Auto ->
              let in_file_mly =
                (Filename.concat indirname base)^".mly" in
              if Sys.file_exists in_file_mly then
                maketitle out_chan base in_file_mly (Ext ".mly")
              else begin
                let in_file_mll =
                  (Filename.concat indirname base)^".mll" in
                if Sys.file_exists in_file_mll then
                  maketitle out_chan base in_file_mll (Ext ".mll")
                else begin
                  let in_file_mli =
                    (Filename.concat indirname base)^".mli" in
                  if Sys.file_exists in_file_mli then begin
                    maketitle out_chan base in_file_mli Intf;
                    state := Skip;
                    let in_file_p4 =
                      (Filename.concat indirname base)^".p4" in
                    if Sys.file_exists in_file_p4 then begin
                      maketitle out_chan base
                        in_file_p4 Impl
                    end else
                end else

```

```

        let in_file_ml =
          (Filename.concat indirname base)^".ml" in
        if Sys.file_exists in_file_ml then
          maketitle out_chan base
          in_file_ml Impl
      end else begin
        let in_file_p4 =
          (Filename.concat indirname base)^".p4" in
        if Sys.file_exists in_file_p4 then begin
          maketitle out_chan base
          in_file_p4 Standalone
        end else begin
          let in_file_ml =
            (Filename.concat indirname base)^".ml" in
          if Sys.file_exists in_file_ml then begin
            maketitle out_chan base
            in_file_ml Standalone
          end else begin
            print_endline ((Sys.argv.(0))
                          ^": Invalid argument.");
            Arg.usage speclist usage_msg;
            exit 1
          end
        end
      end
    end
  end
end
| m -> maketitle out_chan base !file m
end;
print_trailer out_chan
end;;

Printexc.catch main ();

```

5 History

1.2.1	compilation for ocaml 3.08 (stream syntax with camlp4)
1.2	bug correction: <code>_</code> escaping in title when using <code>-main</code>
1.1	bug correction: definition inside a string
1.0	first public release

6 Acknowledgments

I would like to thank here Paul Feautrier for a preliminary version of this program, Denis Barthou, Philippe Marquet and Julien Soula for their beta-testing and of course Luc Maranget for his lightning fast L^AT_EX to HTML conversion program, HEV^EA.