

# erl2latex: Literal Erlang Programming

Ulf Wiger <ulf@wiger.net>

March 14, 2009

Copyright (c) 2008 Ulf Wiger, John Hughes<sup>1</sup>

## 1 Introduction

This module converts an Erlang source file to latex. The latex file can then be converted to e.g. PDF, using pdf<sub>l</sub>atex or similar tool.

The idea of ‘literal Erlang programming’ is that the source and comments should read as a good paper. Unlike XML markup, Latex markup is also fairly unobtrusive when reading the source directly.

See the Makefile for hints on how to integrate erl2latex into your own build system. You can call the emktex script using a symbolic link to the original script, which is located in the erl2latex/src directory.

```
-module(erl2latex).  
  
-export([file/1, file/2]).
```

## 2 file/[1,2]

The interface is:

file(Filename [, Options]) -> ok | {error, Reason}

Options can be specified either when calling file/[1,2], or by adding an attribute, -erl2latex(Options), in the source code. The attribute will not be

---

<sup>1</sup>The MIT License

Copyright (c) 2008 Ulf Wiger <ulf@wiger.net>, John Hughes <john.hughes@quviq.com>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

included in the latex output. Options given in the function call will shadow options embedded in the source code.

mode: ‘normal’ or ‘included’. If mode is ‘included’, preamble and document begin and end markers are removed if found.

```
-spec file/1 :: (Filename::string()) -> ok | {'error',atom()}.

file(F) ->
    file(F, []).
```

The following options control the Latex production: - {mode, included} can be used when making a single document out of several erlang modules and Latex source documents. This has not been tested yet, and probably needs more work before it can be used. - ‘escape\_mode’ controls how special Latex characters are handled. See more below. - ‘source\_listing’ can be used to define custom formatting of the Erlang source code.

```
-type option() :: {documentclass, none | auto | string()}
                  | {mode, normal | included}
                  | {escape_mode, auto | never | always}
                  | {source_listing, auto | string()}
                  | {output, string | binary | file}.

-spec file/2 :: (Filename::string(), [option()]) ->
    {ok,string()} | {'error',atom()}.

file(F, Options) ->
    case file:read_file(F) of
        {ok, Bin} ->
            Target = latex_target(F, Options),
            output(convert_to_latex(Bin, Options), Target);
        Err ->
            Err
    end.
```

### 3 Conversion to Latex

Below is the actual conversion function. We separate comments from code, and convert each block to latex separately. We then insert a preamble, if not already present, or insert a small formatting macro for the source code (if not already defined).

```
convert_to_latex(Bin, Options0) ->
    Parts0 = split_input(binary_to_list(Bin)),
    {Parts1, Embedded_options} = embedded_options(Parts0),
    Parts = rearrange_if_escript(Parts1),
    Opts = Options0 ++ Embedded_options,
    Mode = proplists:get_value(mode, Opts, normal),
    case lists:flatten([convert_part(P, Opts) || P <- hide_show(Parts)]) of
        "\\document" ++ _ = Latex0 ->
            {Preamble,Doc} = get_preamble(Latex0),
```

```

[[[Preamble,
    source_listing_setup(Preamble),
    "\\begin{document}\\n" || Mode == normal],
    Doc,
    end_doc(Mode)]];
Latex0 ->
[[[default_preamble(Opts, Mode),
    "\\begin{document}\\n" || Mode == normal],
    Latex0,
    end_doc(Mode)]
end.

```

Code can be excluded from the document using the following notation:  
`%% @hide ... jcomments and/or codej ... %% @show`

The `@hide` and `@show` instructions must be alone on their respective comment lines (leading and trailing whitespace doesn't matter). Everything between the `@hide` and the `@show` will be excluded from the generated document.

```

hide_show(Parts) ->
{_, Parts1} =
    lists:foldl(
        fun({code,_}=C, {Mode, Acc}) ->
            case Mode of
                hide -> {hide, Acc};
                show -> {show, [C|Acc]}
            end;
        ({comment,Ls}, {Mode, Acc}) ->
            {Mode1, Ls1} = hide_show1(Ls, Mode),
            case Ls1 of
                [] -> {Mode1, Acc};
                [_|_] -> {Mode1, [{comment, Ls1} | Acc]}
            end
        end, {show, []}, Parts),
    lists:reverse(Parts1).

```

`hide_show1/2` traverses the lines of a comment part.

```

hide_show1(Ls, Mode) ->
{Mode1, Ls1} = lists:foldl(
    fun({_, "@show"}, {_, Acc}) -> {show, Acc};
    ({_, "@hide"}, {_, Acc}) -> {hide, Acc};
    (_L, {hide, Acc}) -> {hide, Acc};
    (L, {show, Acc}) -> {show, [L|Acc]}
    end, {Mode, []}, Ls),
{Mode1, lists:reverse(Ls1)}.

```

If a preamble is present, the `\begindocument` entry must also be present. This is how we know where the preamble ends. This function is called on the latex output after processing.

```

-spec get_preamble/1 :: (Str::string()) -> {string(), string()}.

get_preamble(Str) ->
    get_preamble(Str, []).

```

```

get_preamble("\\begin{document}" ++ Rest, Acc) ->
  {lists:reverse("\n" ++ Acc), Rest};
get_preamble([H|T], Acc) ->
  get_preamble(T, [H|Acc]).

```

The following functions output the default latex preamble and document end marker.

```

default_preamble(_Options, included) ->
  "";
default_preamble(Options, normal) ->
  [Doc_class,Src_listing] =
    [proplists:get_value(P,Options) ||
     P <- [documentclass, source_listing]],
  document_class(Doc_class) ++ source_listing_setup(Src_listing,"").

document_class(Opt) ->
  if Opt==auto; Opt==undefined ->
    "\\documentclass[a4paper,12pt]{article}\n";
    Opt==none -> "";
    is_list(Opt) -> Opt
  end.

source_listing_setup(Opt,Preamble) ->
  case regexp:first_match(Preamble, "begin{mylisting}") of
    {match,_,_} ->
      [];
    nomatch ->
      source_listing_setup(Opt)
  end.

source_listing_setup(undefined) ->
  source_listing_setup
    ("\\setlength{\\leftmargin}{1em}"
     "\\item\\scriptsize\\bfseries");
source_listing_setup(Str) when is_list(Str) ->
  ("\\newenvironment{mylisting}\n"
   "{\\begin{list}{\n"
   ++ Str
   ++ (\n"
     "\\end{list}}\n"
   "\n").

end_doc() ->
  "\n\\end{document}\n".

end_doc(included) ->
  "";
end_doc(normal) ->
  end_doc().

```

In this function, we wrap the different ‘source’ and ‘comment’ blocks appropriately. The weird-looking split between string parts in the ‘code’ block is to keep pdflatex from tripping on what looks like the end of the verbatim block when converting its own source to pdf.

The comments can either be processed as latex or non-latex. The difference lies in the handling of the special characters, { } \_ %. By default, pdflatex will look for start and end markers for the latex parts. These are @\_ to start a multi-line block and \_@ to end it. @\_ must be the first symbol on its line, and \_@ must be the last. It is perfectly ok for them to be the *ononly*off symbols on their respective lines. Whitespace before and after is ignored.

The @\_\_ marker denotes a single line of Latex formatting. It must be the first non-whitespace symbol on its line (apart from the initial %, that is.)

Comments that are not identified by these markers will be treated as normal text (possibly commented-out source code), and the special Latex control characters will be automatically escaped.

This behaviour can be modified using the option ‘escape\_mode’. ‘auto’ is the default, described above, ‘never’ means that all comments will be treated as valid Latex, and ‘always’ means that all comments will be treated as non-Latex, and all control characters will be escaped.

In the convert\_part/2 function, the work to classify the comments as Latex or non-Latex is always done. This has the effect of always removing the start and end markers, if there are any. If the escape mode is ‘always’ or ‘never’, the blocks will be re-cast into the expected type.

```
convert_part({code,[]}, _Opts) -> [];
convert_part({code,Lines}, _Opts) ->
  ["\\begin{mylisting}\\n"
   "\\begin{verbatim}\\n",
   [expand(L) || L <- normalize(unwrap(Lines))],
   "\\\" \"end{verbatim}\\n"
   "\\end{mylisting}\\n\\n"];
convert_part({comment,Lines}, Opts) ->
  CParts = classify_clines(Lines),
  CParts1 = case escape_mode(Opts) of
    auto -> CParts;
    never -> [{latex, Ls} || {_, Ls} <- CParts];
    always -> [{comment, Ls} || {_, Ls} <- CParts]
  end,
  Latex = lists:map(
    fun({latex, Ls}) ->
      [[L,"\\n"] || {_,L} <- Ls];
    ({comment, Ls}) ->
      [[escape(L), "\\n"] || {_,L} <- Ls]
  end, CParts1),
  [Latex,"\\n"].

escape_mode(Opts) ->
  proplists:get_value(escape_mode, Opts, auto).

unwrap(Ls) ->
  [L || {_,L} <- Ls].

normalize(["\\n", "\\n"|T]) -> normalize(["\\n"|T]);
normalize([H|T]) -> [H|normalize(T)];
normalize([]) -> [].
```

```

classify_clines([Ln, "@_" ++ L} | Ls]) ->
    [{latex, [{Ln, strip_empty_chars(L)}]} | classify_clines(Ls)];
classify_clines([Ln, "@" ++ L} | Ls]) ->
    Ls1 = case strip_empty_chars(L) of
        [] -> Ls;
        [_|_] = L1 -> [{Ln, L1}|Ls]
    end,
    {LLs, Rest} = collect_latex(Ls1, Ln),
    [{latex, LLs} | classify_clines(Rest)];
classify_clines([Ln, L} | Ls]) ->
    [{comment, [{Ln, escape(L)}]} | classify_clines(Ls)];
classify_clines([]) ->
    [].

collect_latex(Ls, Ln) ->
    collect_latex(Ls, [], Ln).

collect_latex([_, "@_" ++ _} = L | _], _Acc, _PrevLn) ->
    erlang:error({nested_latex_brackets, L});
collect_latex([_, "@" ++ Rest} = L | Ls], Acc, _PrevLn) ->
    case strip_empty_chars(Rest) of
        [] ->
            {lists:reverse(Acc), Ls};
        [_|_] ->
            erlang:error({text_after_latex_end_bracket, L})
    end;
collect_latex([Ln, L} | Ls], Acc, _PrevLn) ->
    case strip_space(lists:reverse(L)) of
        "@_" ++ RevL ->
            {lists:reverse([Ln, lists:reverse(RevL)} | Acc]), Ls};
        _ ->
            collect_latex(Ls, [Ln, L} | Acc], Ln)
    end;
collect_latex([], _Acc, PrevLn) ->
    erlang:error({missing_latex_end_bracket, PrevLn}).

```

The `expand(Line)` function expands tabs for better formatting.

```

expand(Str) ->
    expand_tabs(Str).

expand_tabs(Xs) ->
    expand_tabs(0, Xs).

expand_tabs(_N, []) ->
    [];
expand_tabs(N, [$\t|Xs]) ->
    N1 = 8*(N div 8) + 8,
    [$\s || _ <- lists:seq(N, N1)] ++ expand_tabs(N1, Xs);
expand_tabs(N, [X|Xs]) ->
    [X|expand_tabs(N+1, Xs)].

```

A potential problem is when the programmer has commented out code. This code is then likely to contain e.g. underscores, which will probably trip up the Latex to pdf conversion. Trying to detect whether the commented text is Erlang code is tricky, since it's rather likely that it's not a syntactically complete expression. We take the approach that we scan the code

for unescaped underscores, and then deal with how to deal with "real LaTeX underscores" later (let's assume to begin with that they are not that common).

```
escape("\\\" ++ [H|Rest]) ->
  "\\\" ++ [H|escape(Rest)];
escape("_" ++ Rest) ->
  "\\_" ++ escape(Rest);
escape([H|T]) ->
  [H|escape(T)];
escape([]) ->
  [].
```

Following edoc convention, comments are excluded if the first non-space character following the leading string of % is another %, for example:  
%% % This comment will be excluded.

```
strip_comment(C) ->
  C1 = strip_percents(C),
  case string:strip(C1, left) of
    "%" ++ _ -> "";
    Stripped ->
      Stripped
  end.

strip_percents("%" ++ C) -> strip_percents(C);
strip_percents(C)          -> C.
```

## 4 Utility Functions

```
split_input(Txt) ->
  [{T1,Ls} ||
    {T1,Ls} <-
      [{T,strip_empty_lines(L1)} ||
        {T,L1} <- group([wrap(L) || L <- line_nos(lines(Txt))])],
    Ls /= []].

lines(Str) ->
  lines(Str, []).

lines("\n" ++ Str, Acc) ->
  [lists:reverse([$\\n|Acc]) | lines(Str,[])];
lines([H|T], Acc) ->
  lines(T, [H|Acc]);
lines([], Acc) ->
  [lists:reverse(Acc)].

line_nos(Ls) ->
  line_nos(Ls, 1).

line_nos([L|Ls], N) ->
  [{N,L}|line_nos(Ls, N+1)];
line_nos([], _) ->
  [].
```

```

wrap({N, "%" ++ S}) ->
  {comment, N, strip_empty_chars(strip_comment(S))};
wrap({N, S}) ->
  {code, N, S}.

strip_empty_lines(Ls) ->
  Strip = fun(Ls1) ->
    lists:dropwhile(fun({_,L}) ->
      strip_empty_chars(L) == []
    end, Ls1)
  end,
  lists:reverse(Strip(lists:reverse(Strip(Ls)))).

strip_empty_chars(L) ->
  lists:reverse(strip_space(lists:reverse(strip_space(L)))).

```

Remove leading empty lines, even if they contain whitespace.

```

strip_space(L) ->
  lists:dropwhile(fun(C) when C==$\s; C==$\t; C==$\n -> true;
    (_) -> false
  end, L).

group([T,N,C|Tail]) ->
  {More,Rest} = lists:splitwith(fun({T1,_N1,_C1}) -> T1 == T end, Tail),
  [{T,[N,C]|[_N1,C1] || {_N1,C1} <- More}] | group(Rest)];
group([]) ->
  [].

latex_target(F, Options) ->
  case proplists:get_value(output, Options, file) of
    file ->
      Target_base = strip_extension(F) ++ ".tex",
      Outdir = proplists:get_value(outdir, Options, filename:dirname(F)),
      {file, filename:join(Outdir, Target_base)};
    T when T==binary;
      T==string ->
      T
  end.

strip_extension(F) ->
  case regexp:split(F, "\.[a-z]+$") of
    {ok, [F1,[]]} ->
      F1
  end.

output(Data, string) ->
  {ok, binary_to_list(list_to_binary(Data))};
output(Data, binary) ->
  {ok, list_to_binary(Data)};
output(Data, {file, F}) ->
  case file:write_file(F, list_to_binary(Data)) of
    ok ->
      {ok, F};
    Err ->
      Err
  end.

embedded_options(Parts) ->
  lists:mapfoldl(

```



```

    fun({code,Ls}=Part,Acc) ->
        case scan_for_opts(Ls) of
            none -> {Part,Acc};
            {Opts,Ls1} -> {{code,Ls1}, Acc ++ Opts}
        end;
    (Other, Acc) ->
        {Other, Acc}
    end, [], Parts).

scan_for_opts(Ls) ->
    scan_forms(Ls, [], []).

scan_forms([], Opts, Acc) ->
    case Opts of
        [] -> none;
        [_|_] -> {Opts, Acc}
    end;
scan_forms(Ls, Opts0, Acc) ->
    case scan_tokens(Ls) of
        {{ok,[{'-',_},{atom,_,erl2latex},{'(',L}|Ts],_}, Used, Rest} ->
            case erl_parse:parse_term([{'(',L}|Ts]) of
                {ok, Opts} -> scan_forms(Rest, Opts0++Opts, Acc);
                {error,_} -> scan_forms(Rest, Opts0, Acc ++ Used)
            end;
        {{eof,_,_}, Used, []} ->
            case Opts0 of
                [_|_] -> {Opts0, Acc ++ Used};
                [] -> none
            end;
        {_, Used, Rest} ->
            scan_forms(Rest, Opts0, Acc ++ Used)
    end.

scan_tokens([Ln,Str]=L|Ls) ->
    scan_tokens(erl_scan:tokens([],Str,Ln), Ls, [L]).

scan_tokens({done,Result,Leftover_chars},Rest,Used) ->
    Rest1 =
        case Leftover_chars of
            [] -> Rest;
            [_|_] ->
                Ln = element(1,hd(Used)),
                [{Ln,Leftover_chars}|Rest]
        end,
    {Result, lists:reverse(Used), Rest1};
scan_tokens({more, Cont}, Ls, Used) ->
    case Ls of
        [] ->
            {{eof,1}, lists:reverse(Used), []};
        [{Ln,Str}=L|Rest] ->
            scan_tokens(erl_scan:tokens(Cont, Str, Ln), Rest, [L|Used])
    end.

rearrange_if_escript([code, [{_, "#!" ++ _} = Head]],
    {comment, [{Nx, "!" ++ _} = Xtra]}],
    {comment, _} = Cmt|Rest) ->
    Code = {code, [Head, {Nx, "%%" ++ Xtra}]},

    [Cmt, Code | Rest];
rearrange_if_escript([code, [{_, "#!" ++ _}|_] = Head,
    {comment, _} = Cmt|Rest]) ->
    [Cmt, Head | Rest];

```

```
rearrange_if_escript(Other) ->  
  Other.
```