

# The SimpleLit System for Literate Programming

Andrew Rock  
School of Information and Communication Technology  
Griffith University  
Nathan, Queensland, 4111, Australia  
a.rock@griffith.edu.au

January 24, 2008

## Abstract

This document is a description of the SimpleLit program that forms the basis for a very simple system for literate programming in any language or mix of languages. In essence the program allows one source file to specify the contents of many files of various types.

## 1 Introduction

Literate programming is a discipline that results in well documented programs that stay well documented as the programs evolve. Ordinary in-source comments are inadequate if the program is best described using mathematics or diagrams. Separate design or implementation documentation can be put aside and falls out of synch with the programs. The inclusion of documentation to be typeset with a powerful tool like  $\text{\LaTeX}$  in the source file with the program encourages the programmer to maintain the documentation as a normal part of the program maintenance.

Elaborate literate programming tools such as `WEB` [1, 2] are overkill. `WEB`'s aptly named `tangle` tool is not appropriate for languages such as Haskell that use the code's layout to define the code's structure. The simple mechanisms provided by Haskell-like languages for literate programming are not flexible enough. The SimpleLit tool described and implemented below is a nice middle ground. It is deliberately clean and simple and contains very few optimizations or adaptations specific to any markup or programming languages. It could be used, as here, to combine Haskell and  $\text{\LaTeX}$ , or HTML and Java, or all four.

## 2 SimpleLit Program Design

### 2.1 Definitions

The following definitions describe the structure of a source file to be processed with SimpleLit.

**Source:** A source file is a sequence of lines. Line endings should be whatever combination of control characters is standard on the current platform: `CR` on a Macintosh; `LF` on UNIX; or both on a PC.

**LitLine:** A LitLine is a line that has `[` in the first column. There may be a closing `]` on the same line. Either way, the LitLine ends at the end of the line. LitLines contain directives to the SimpleLit tool and/or comments that are never output.

**PlainLine:** A PlainLine is any line that is not a LitLine, that is, it starts with anything other than a `[`. PlainLines are written to the various destinations by the SimpleLit tool. A line that has `][` in the first two columns is by the above definition a PlainLine. The leading `]` will be removed before the line is written. This escape permits an output PlainLine to begin with `[`. Just in case anyone really wants to start a PlainLine with `][`, the input sequence `]][` will be output as a PlainLine starting with `][`.

**LitDirective:** Any text on a LitLine between the `[` in the first column and a closing end of line or `]` is a directive to the SimpleLit program.

**LitComment:** Any text that follows the `]` on a LitLine is a LitComment. It will be ignored. If the LitDirective is empty, the whole line is effectively a comment.

**Destinations:** PlainLines from the Source are printed to one or more destinations including standard output and named files. The named files form a set. Only if that set is empty, will lines instead be printed to standard output. Standard output is the initial destination.

**Line-number-preserved Destinations:** A set of output files to which are written the same number of lines as the Source. Blank lines are output to these files for each line in the Source for which they are open, but not in the current set Destinations.

## 2.2 Directive Descriptions

Within the `[` and newline or `]` that delimit a LitDirective there may be zero or more directives. Each is processed in order. Multiple directives in the one LitDirective should be separated by spaces or tabs. The directives are shown in table 1. Anything other than a recognized directive occurring in a LitDirective is ignored, apart from issuing a warning.

# 3 Using the SimpleLit Tool

## 3.1 Using the compiled tool

If SimpleLit has been compiled, use the command:

```
SimpleLit source-file-names...
```

If no *source-file-names* are provided, then standard input is read instead. Each source file is processed independently. It is possible for the output from one source to clobber the output from a previous source.

<code>+filename</code>	Add the named file to the current set of destinations. If the file is not currently open, open it for writing.
<code>++filename</code>	Add the named file to the current set of destinations. If the file is not currently open, open it for appending, that is, don't clobber its current contents. If the file was already open, issue a warning. (In most cases <code>+filename</code> is preferred.)
<code>-filename</code>	Remove the named file from the current set of destinations. Warn if the file is not in the current set of destinations. Leave the file open for more output.
<code>--filename</code>	Remove the named file from the current set of destinations. Warn if the file is not in the current set of destinations. Close the file. (In most cases <code>-filename</code> is preferred, unless you want to open the file again in the same run.)
<code>--*</code>	Remove all named files from the set of destinations, making standard output the only destination. Leave all the files open for more output. (There is no <code>--*</code> directive as it would be ambiguous as to which files had been closed.)
<code>&lt;filename</code>	Include the named file in the source. This mechanism can be used a poor man's macro facility. Warning: This will output extra lines to Line-number-preserved Destinations, consequently these features are incompatible.
<code>#filename</code>	Include the named file in the set of Line-number-preserved Destinations. This opens the file if it is not already open for writing. This directive should be issued on the <i>first</i> line of the Source to ensure that blank lines are written from the start of the file.

Table 1: The SimpleLit directives.

## 3.2 Using Hugs

The source for this document is itself a valid literate Haskell script that can be loaded into Hugs and executed. Evaluation of the expression

```
run "source-file-name"
```

will process the named source file.

## 3.3 Bootstrapping

To compile SimpleLit from its single literate source, `SimpleLit.lhs`, use this procedure.

1. Type this command to compile the tool, using GHC (Glasgow Haskell Compiler):

```
ghc SimpleLit.lhs -o SimpleLit
```

2. Type this command to extract the make file and L<sup>A</sup>T<sub>E</sub>X documentation:

```
SimpleLit SimpleLit.lhs
```

3. Use the extracted Makefile to build the compiled tool (if you haven't already) and the PDF documentation (requires make, pdf<sub>l</sub>atex and bibtex):

```
make
```

## 3.4 Tips

### 3.4.1 Emacs

Peter Simons ([simons@cryp.to](mailto:simons@cryp.to)) recommends MMM-Mode for Emacs (Multi-Mode-Mode), which is available at SourceForge, to support multiple syntax modes within the one file.

## 4 SimpleLit Program Implementation

SimpleLit is implemented with one Haskell module.

```
module Main(main) where
```

```
import Prelude hiding (error); import System; import IO
import List
```

main processes each of the command line arguments or standard input if there are none.

```
main :: IO ()
main = do
  args <- getArgs
  if null args
    then processStdin
    else mapM_ processFile args
  exitWith ExitSuccess
```

processStdin processes standard input.

```
processStdin :: IO ()
processStdin = do
  notify 0 "Reading standard input."
  text <- getContents
  processLines [] [] (zip [1..] (lines text)) []
  notify 0 "Done reading standard input."
```

processFile *path* processes the file named *path*.

```
processFile :: FilePath -> IO ()
processFile path = do
  notify 0 ("Reading source file: " ++ path)
  text <- catch (readFile path) (\e -> do
```

```

        error 0 "File can not be read."
        return ""
    )
    processLines [] [] (zip [1..] (lines text)) []
    notify 0 ("Done reading source file: " ++ path)

```

run, the preferred Hugs entry point, is just an alias for processFile.

```

run :: FilePath -> IO ()
run = processFile

```

processLines *phs hs nls* processes the remaining lines of text (paired with their line numbers) *nls*, using the current set of destination file handles *hs*, the lookup table mapping file paths to handles *phs*, and the set of line-number-preserved file handles, *lhs*.

```

processLines :: [(FilePath, Handle)] -> [Handle] ->
  [(Int,String)] -> [Handle] -> IO ()
processLines phs _ [] _ = mapM_ (hClose . snd) phs
processLines phs hs ((n,l):nls) lhs = case l of
  '[' : l'      -> directive phs hs (words (takeWhile
    (/= ']') l')) lhs
  ']' : ']' : l' -> plainLine (']' : l')
  ']' : ']' : ']' : ']' : l' -> plainLine (']' : ']' : ']' : l')
  -             -> plainLine l
where
plainLine :: String -> IO ()
plainLine l = do
  if null hs
    then putStrLn l
    else mapM_ (\h -> hPutStrLn h l) hs
  mapM_ (\h -> hPutStrLn h "") (lhs \\ hs)
  processLines phs hs nls lhs
directive :: [(FilePath, Handle)] -> [Handle] -> [String] ->
  [Handle] -> IO ()
directive phs hs [] lhs = do
  mapM_ (\h -> hPutStrLn h "") lhs
  processLines phs hs nls lhs
directive phs hs (d:ds) lhs = case d of
  "-*"          -> directive phs [] ds lhs
  '+' : '+' : c : cs -> case lookup (c:cs) phs of
    Just h -> do
      warning n ((c:cs) ++
        " was already open and can't be \
        \opened for appending.")
      directive phs (nub (h:hs)) ds lhs
    Nothing -> do
      h <- openFile (c:cs) AppendMode
      directive (((c:cs), h) : phs) (h:hs) ds lhs
  '+' : c : cs -> case lookup (c:cs) phs of
    Just h ->

```

```

        directive phs (nub (h:hs)) ds lhs
Nothing -> do
    h <- openFile (c:cs) WriteMode
    directive (((c:cs), h) : phs) (h:hs) ds lhs
'->' : '->' : c : cs -> case lookup (c:cs) phs of
    Just h -> do
        hClose h
        directive (delete ((c:cs),h) phs) (delete h hs) ds
            (delete h lhs)
Nothing -> do
    warning n ((c:cs) ++
        " was not a current destination.")
    directive phs hs ds lhs
'->' : c : cs -> case lookup (c:cs) phs of
    Just h -> directive phs (delete h hs) ds lhs
Nothing -> do
    warning n ((c:cs) ++
        " was not a current destination.")
    directive phs hs ds lhs
'<' : c : cs -> do
    notify n ("Including source file: " ++ (c:cs))
    t <- catch (readFile (c:cs)) (\e -> do
        error 0 "Included file can not be read."
        return "")
    )
    processLines phs hs (zip [1..] (lines t) ++
        [(n, '[' : unwords (('\'0':c:cs):ds)]) ++ nls) lhs
'#>' : c : cs -> case lookup (c:cs) phs of
    Just h -> directive phs hs ds (nub (h : lhs))
Nothing -> do
    h <- openFile (c:cs) WriteMode
    directive (((c:cs), h) : phs) hs ds (h:lhs)
'\0' : c : cs -> do
    notify 0 ("Done including source " ++ (c:cs))
    directive phs hs ds lhs
-
    error n ("Unknown directive: " ++ d)
    directive phs hs ds lhs

```

Use `notify n m` to notify the user of an event at line `n` with message `m`. Use `warning n m` to issue warnings. Use `error n m` to issue fatal error messages.

```

notify, warning, error :: Int -> String -> IO ()
notify n msg = message "----" "" n msg
warning n msg = message "+++" " Warning" n msg
error n msg = message "###" " Error" n msg

message :: String -> String -> Int -> String -> IO ()
message flag1 flag2 n msg = hPutStrLn stderr $
    "[" ++ flag1 ++ (if n > 0 then show n else "") ++
    "]" SimpleLit" ++ flag2 ++ ": " ++ msg

```

## 5 Makefile

This make file compiles the SimpleLit tool and typesets this document.

### 5.1 Fake targets

The target `all` builds the program and documentation in PDF.

```
all : SimpleLit SimpleLit.pdf
```

The target `clean` removes some intermediate files. The target `CLEAN` removes more intermediate files.

```
clean :
    rm *.hi *.o *.aux *.log *.out *.lm *.brf *.bbl *.blg
```

```
CLEAN : clean
    rm *.tex *.bib
```

### 5.2 Options

```
GHC = ghc
```

### 5.3 Real targets

This dependency shows how SimpleLit is used to produce the L<sup>A</sup>T<sub>E</sub>X and Haskell sources and the Makefile.

```
SimpleLit.tex SimpleLitHB.tex Makefile : SimpleLit.lhs
    SimpleLit SimpleLit.lhs
```

This produces the PDF documentation.

```
SimpleLit.pdf : SimpleLit.tex
    pdflatex SimpleLit.tex
    bibtex SimpleLit
    pdflatex SimpleLit.tex
    pdflatex SimpleLit.tex
```

This compiles the tool.

```
SimpleLit : SimpleLit.lhs
    $(GHC) SimpleLit.lhs -o SimpleLit
```

This typesets my handbook extract.

```
h : SimpleLitHB.tex
    pdflatex SimpleLitHB.tex
    pdflatex SimpleLitHB.tex
    open SimpleLitHB.pdf
```

## References

- [1] Donald Knuth. Literate programming. *The Computer Journal*, 27:97–111, 1984. [1](#)
- [2] Donald E. Knuth. *Literate Programming*, chapter 4. Number 27 in CSLI Lecture Notes. Center for the Study of Language and Information, 1991. [1](#)