

# Illiterate Programming\*

Asher Hoskins

23 December 2004

## Contents

<b>1 Overview</b>	<b>1</b>
<b>2 Packages</b>	<b>1</b>
<b>3 Global Variables</b>	<b>2</b>
3.1 Buffers . . . . .	2
3.2 Other Globals . . . . .	2
<b>4 Configuration</b>	<b>2</b>
4.1 Built-in Defaults . . . . .	2
4.2 Configuration Files . . . . .	3
4.3 Command Line Options . . . . .	3
4.4 Get Input and Output File Names . . . . .	4
<b>5 Main Loop</b>	<b>5</b>
<b>6 Read Configuration File</b>	<b>6</b>
<b>7 Comment Sorting</b>	<b>8</b>
7.1 Calculate comment indent . . . . .	8
7.2 Remove comment characters from lines . . . . .	8
7.3 Sort comments . . . . .	9
<b>8 Add a Line to a Buffer</b>	<b>10</b>
<b>9 Add a Line to the Code Buffer</b>	<b>11</b>
<b>10 Expand Shorthand in a Line</b>	<b>11</b>
<b>11 Expand Metadata</b>	<b>11</b>
<b>12 Dump Buffer</b>	<b>12</b>
<b>13 Dump the Code Buffer</b>	<b>13</b>
<b>14 Dump the Crossed-Out Buffer</b>	<b>14</b>

---

\*File `illit`, version 0.0.5.

<b>15 Dump Non-Indented Text</b>	<b>14</b>
<b>16 Dump Indented Text</b>	<b>15</b>
<b>17 Trim Blank Leading &amp; Trailing Lines</b>	<b>15</b>
<b>18 Output the Formatted Buffer in a Template</b>	<b>15</b>
18.1 Construct the Template File Name . . . . .	16
18.2 Redirect the Output . . . . .	16
18.3 Scan the Template . . . . .	16
<b>19 Output Metadata</b>	<b>17</b>
<b>20 Quote <math>\LaTeX</math> Special Characters</b>	<b>17</b>

## 1 Overview

illit is an attempt to produce a literate programming system for modern programmers. It produces high quality  $\LaTeX$  output, allows the programmer to structure and index their program to aid source-code understanding, doesn't require the source code to be pre-processed before it can be interpreted or compiled, and, most importantly of all for the majority of programmers who spend most of their time looking at a screen rather than a printout, doesn't require the source to be cluttered up with ugly escape codes. It tries to work using existing commenting conventions as much as possible and so fits around the programmer rather than forcing the programmer to fit around it.

- TODO:** *Allow template to set how metadata output? Use for change history?*
- TODO:** *Allow tag/shorthand continuation over multiple lines with backslash?*
- TODO:** *Search for template in current dir if not found in template dir.*
- TODO:** *Make template dir arg a colon-separated list.*
- TODO:** *Make illit  $\LaTeX$  package to load required pkgs automatically.*

## 2 Packages

```

35 use strict;
    Expand tabs to spaces.
37 use Text::Tabs;
    Pretty printing of data structures while debugging.
use Data::Dumper;
    Handle command line options.
41 use Getopt::Long qw(:config no_ignore_case);
    Import the $HOME environment variable;
43 use Env 'HOME';

```

## 3 Global Variables

### 3.1 Buffers

Buffers for code and comment lines along with a state variable to keep track of which one we last used.

```
50 my $lastbuf = 0;
51 my @code = ();
52 my @noindent = ();
53 my @crossedout = ();
54 my @indent = ();
```

Keep track of what line the current bit of code started at.

```
56 my $codestart = 0;
```

The buffer for the formatted code.

```
58 my @formatted = ();
```

### 3.2 Other Globals

The name of the program.

```
62 my $myname = "illit";
63
64 {
```

## 4 Configuration

This program can be configured in five different places:

1. The built-in data structures.
2. The the `$HOME/.illitrc` or `$HOME/.illit/.illitrc` configuration files.
3. A `.illitrc` file in the current directory.
4. The command line.
5. The source file that is being formatted.

Configuration options set in each point in this list can be overridden by options set in the points following it.

### 4.1 Built-in Defaults

```
80 my $conf = {
    Default to Perl for now.
82     language => "perl",
    Template file and directory.
```

```

84     tempdir => ".",
85     template => "default_tmpl.tex",

    Language-specific settings.

87     lang => {
88         c => {
89             cline => '//',
90             cstt => '/*',
91             cign => '*',
92             cend => '*/',
93             tab => 4
94         },
95         perl => {
96             cline => '#',
97             cstt => '',
98             cign => '#',
99             cend => '',
100            tab => 4
101        },
102        tex => {
103            cline => '%',
104            cstt => '',
105            cign => '%',
106            cend => '',
107            tab => 4
108        },
109    },

    Metadata.

111    meta => {}
112 };

```

## 4.2 Configuration Files

Test if there's a global configuration file. These can be in either of two places. If they exist in both places then only the first is read.

```

118 if (-r "$HOME/.illitrc") {
119     configfile("$HOME/.illitrc", $conf);
120 }
121 elseif (-r "$HOME/.illit/.illitrc") {
122     configfile("$HOME/.illit/.illitrc", $conf);
123 }

```

Test for a configuration file in the current directory. This may contain overrides to the global file but does not replace it.

```

126 if (-r ".illitrc") {
127     configfile(".illitrc", $conf);
128 }

```

## 4.3 Command Line Options

Use `Getopt::Long` to parse command line options.

```

133 my $usertab;
134 GetOptions(
135     'tempdir|d=s' => \$conf->{tempdir},
136     'template|t=s' => \$conf->{template},
137     'language|l=s' => \$conf->{language},
138     'tabstop|s=i' => \$usertab,
139     'meta|D|m=s' => $conf->{meta}
140 );

```

Check that the options make sense. There's no need to do this until after all the options have been defined (config file and command line).

```

143 unless (defined $conf->{lang}->{$conf->{language}}) {
144     die "myname: Unknown language \"\$conf->{language}\".\n";
145 }

```

The `tabstop` option is a little complicated since it overrides a per-language setting. We put it in a temporary variable until all the command line options (which may change the language setting) have been parsed so that we can change the `tabstop` once we know which language it should affect.

```

151 if (defined $usertab) {
152     $conf->{lang}->{$conf->{language}}->{tab} = $usertab;
153 }

```

We don't just check `$usertab` here since an incorrect `tab` may have been set in a configuration file.

```

156 unless ($conf->{lang}->{$conf->{language}}->{tab} >= 0) {
157     die "myname: Tabstops may not be negative.\n";
158 }

```

Show results of default+file+cmdline options.

```

print Dumper($conf);
die;

```

## 4.4 Get Input and Output File Names

```

165 my $infile = "";
166 my $outfile = "";
167 if (scalar(@ARGV) > 0) {
168     $infile = shift;

```

If the `.fi` meta tag has not yet been set (this will generally be set on the command line when this program is being used in a script and fed files from `stdin`) then set it to the file name.

```

172     unless ($conf->{meta}->{fi}) {
173         $conf->{meta}->{fi} = quotetex($infile);
174     }
175 }
176 if (scalar(@ARGV) > 0) {
177     $outfile = shift;
178 }

```

## 5 Main Loop

```
Input line buffer.
185 my $line;

Input line number.
187 my $linenum = 0;

State: 0: not in comment, 1: in comment.
189 my $comment = 0;

Shorthand link for the language-specific configuration.
192 my $lang = $conf->{lang}{$conf->{language}};

Open the input file or set it to STDIN if no filename was given.

195 local *IN;
196 if ($infile) {
197     unless(open(IN, $infile)) {
198         die "$myname: Couldn't open input file \"$infile\".\n";
199     }
200 }
201 else {
202     *IN = *STDIN;
203 }

Loop through the whole file.
206 while ($line = <IN>) {

    Tidy up line: remove EOL character & expand tabs.

208     chomp $line;
209     $tabstop = $lang->{tab};
210     $line = expand($line);
211
212     $linenum++;

    Check for hash-bang initial line. Ignore it.

215     if ($linenum==1 && $line=~/^#!/) {
216         next;
217     }

    If we're not in a comment then check for the start of one or for a single-
    line comment. We don't allow mixed code and comment lines so we just
    need to check for a comment character at the start of the line (ignoring
    any leading whitespace).

223     if (!$comment) {

        Note use of "\Q" (quotemeta) to avoid problems when meta-characters
        appear in the comment delimiters. Use "\E" to end escaping.

227         if ($lang->{cline} && $line=~/^\\s*\\Q$lang->{cline}/) {
228             comment($line, $conf);
229         }
230         elsif ($lang->{cstt} && $line=~/^\\s*\\Q$lang->{cstt}/) {
```

```

                Only set $comment when there's no end-comment string.
232         if ($line!~/\Q$lang->{cend}/) {
233             $comment = 1;
234         }
235         comment($line, $conf);
236     }
237     else {
238         code($line, $linenum);
239     }
240 }

    If we are in a comment then check for the end.
242     else {
243         if ($line=~/\Q$lang->{cend}/) {
244             $comment = 0;
245         }
246         comment($line, $conf);
247     }
248 }
249 }

    Dump any remaining stuff in the buffers.
252 dumpbuffer($lastbuf);

    If we read from a file then close it.
255 if ($infile) {
256     close IN;
257 }

    Output the formatted code in a template.
260 outputtemplate($conf, $outfile);
261
262 }

```

## 6 Read Configuration File (configfile)

Read a configuration file into the configuration hash.

The configuration file is a list of “key=value” pairs, one per line. Blank and comment lines (beginning with “#”) are ignored, as is all whitespace not inside the value. Per-language definitions are put inside a “langdef” block. Meta data may be set using a two word key starting with “meta”.

```

275 sub configfile {
276
277     my ($filename, $conf) = @_;
278     my $line;
279     my $langdef = "";
280     my $key;
281     my $value;
282
283     unless (open CONFIG, $filename) {
284         die "$myname: Could not open configuration file \"$filename\".\n";

```

```

285     }
286
287 while ($line = <CONFIG>) {
288     chomp $line;
289
290     Discard unwanted whitespace (leading, trailing, either side of the equals
291     sign).
292
293     $line =~ s/^\s+//;
294     $line =~ s/\s+$//;
295     $line =~ s/\s+=\s+//;
296
297     Discard blank and comment lines (beginning with "#").
298
299     if ($line =~ /^#/ || $line eq "") {
300         next;
301     }
302
303     Test for the start/end of a language definition.
304
305     if ($line =~ /^langdef/) {
306         Set the language name.
307
308         ($langdef) = $line =~ /^langdef\s*(\S+)$/;
309
310         Set default values for per-language settings.
311
312         $conf->{lang}->{$langdef} = {
313             cline => '',
314             cstt => '',
315             cign => '',
316             cend => '',
317             tab => 4
318         };
319     }
320     elsif ($line eq "enddef") {
321         $langdef = "";
322     }
323
324     Set language specific values.
325
326     elsif ($langdef) {
327         ($key, $value) = $line =~ /^([\^=]+)=(.+)$/;
328         $conf->{lang}->{$langdef}->{$key} = $value;
329     }
330
331     Set metadata.
332
333     elsif ($line =~ /^meta/) {
334         ($key, $value) = $line =~ /^meta\s*(\S+)=(.+)$/;
335         $conf->{meta}->{$key} = $value;
336     }
337
338     Set all other values.
339
340     else {
341         ($key, $value) = $line =~ /^([\^=]+)=(.+)$/;
342         $conf->{$key} = $value;
343     }
344 }

```

```

333
334     close CONFIG;
335
336 }

```

## 7 Comment Sorting (comment)

Decide what sort of comment this line is and pass it off to the appropriate handling routine.

```

344 sub comment {
345
346     my ($line, $conf) = @_;
347     my $lang = $conf->{lang}{$conf->{language}};
348     my $indentsize;

```

### 7.1 Calculate comment indent

Just count initial spaces since tabs have already been expanded. Divide this by `$lang->{tab}` and convert to an integer to give a size in tabstops. The conversion to an integer is necessary to avoid the problem found in C style comments where the first line of a comment (with the correct indent) has a sectioning command and so is treated as a separate to the following block of commentary, which is preceded by “`␣*`” to line the `*` up with the comment starting “`/*`”, resulting in comment indentation that does not match the code it refers to.

```

360     $indentsize = int(length(($line =~ /^(^\s*)/)[0])/$lang->{tab});

```

### 7.2 Remove comment characters from lines

Comment end characters only match at the end of a line (they may be followed by whitespace).

```

365     $line =~ s/\Q$lang->{cend}\E\s*$/;

```

Comment start, single-line comment, and “ignore” characters are only removed from the beginning of lines.

```

368     $line =~ s/^\s*\Q$lang->{cline};//;
369     $line =~ s/^\s*\Q$lang->{cstt};//;
370     if ($lang->{cign} ne '') {
371         $line =~ s/^\s*\Q$lang->{cign}\E+//;
372     }

```

Remove any leading whitespace (to make the comparisons below clearer).

```

374     $line =~ s/^\s*//;

```

### 7.3 Sort comments

Test if the line starts with the code cross-out sequence “X ” or consists of a single “X”.

```
379 if ($line=~^X\s+/ || $line=~^X$/) {
    Remove the cross-out sequence, reindent the line, and add it to the
    crossed-out buffer.
382     $line =~ s/^X$//;
383     $line =~ s/^X\s+//;
384     my $i;
385     for ($i = 0; $i < $indentsize*$lang->{tab}; $i++) {
386         $line = " $line";
387     }
388     addbuffer($line, \@crossedout);
389 }

Parse for metadata (any tag that doesn't begin with s or fn).
391 elsif ($line=~/\.[^s]./ && $line!~/^\.fn/) {
392     metadata($line, $conf);
393 }

Any other comment line is commentary text.
395 else {
    Expand any comment shorthand.
397     $line = shorthand($line);

    If the line starts with a .s?/.fn sectioning tag then expand sectioning
    tags and generate a line of non-indented text.
401     if ($line=~/\.[^s]./ || $line=~^\.fn/) {
        Full stops at the end of section tags are removed.
403         $line =~ s/\.[^s]*$//;

        .sa-.se expand to \seca{}-\sece{}.
405         $line =~ s/\.[^s]([abcde])\s*(.*)/\sec$1\{$2\}/;

        .fn may be used in place of .sa at the start of a function. It should
        contain a single word function name, a "/" separator, and a descrip-
        tion of the function. This is expanded to \func{name}{description}.
        The template can use this to generate index entries for function
        names, etc.
411         $line =~ s/^\.fn\s*

            Match the function name.
413             (\S+)

            Separator.
415             \s*\s*

            Description.
```

```

417         (.*
418         $\backslashfunc\{${1}\}\{${2}\}/x;

        Add it to the @noindent buffer.

420     addbuffer($line, \@noindent);
421 }

    If the indent is zero then generate a line of non-indented text (obvi-
    ously).

424     elsif ($indent==0) {
425         addbuffer($line, \@noindent);
426     }

    All other text is put into an indented block.

428     else {

        The first line of an indented buffer is the indent to use.

430         if (scalar(@indent) == 0) {
431             addbuffer($indent, \@indent);
432         }

        Add the line of text.

434         addbuffer($line, \@indent);
435     }
436 }
437
438 }

```

## 8 Add a Line to a Buffer (addbuffer)

Add a line to a buffer, dumping the previously used buffer if this is the first line in a new buffer.

```

446 sub addbuffer {
447
448     my ($line, $buffer) = @_;

        Add the line to the buffer.

451     push @$buffer, $line;

        If this is the first line in a new buffer then dump the last used one.

454     if ($lastbuf != $buffer) {
455         dumpbuffer($lastbuf);
456         $lastbuf = $buffer;
457     }
458
459 }

```

## 9 Add a Line to the Code Buffer (code)

```
465 sub code {
466
467     my ($line, $linenum) = @_;
        Add the line to the code buffer.
470     push @code, $line;
        If the current buffer is not @code then note the starting line number of the
        code, dump any comments in the buffers, and change the state.
475     if ($lastbuf != \@code) {
476         dumpbuffer($lastbuf);
477         $codestart = $linenum;
478         $lastbuf = \@code;
479     }
480
481 }
```

## 10 Expand Shorthand in a Line (shorthand)

```
487 sub shorthand {
488
489     my ($line) = @_;
        Expand todos, etc.
492     my $tag;
493     foreach $tag ("TODO", "XXX", "FIXME", "NOTE") {
        If there's a colon following the tag then take the rest of the line as an
        argument to it.
496         $line =~ s/($tag)\s*:\s*(.*)/\todo\{$1\}\{$2\}/;
        If it ends the line and has no colon then output an empty tag.
498         $line =~ s/($tag)\s*/\todo\{$1\}\{\}/;
499     }
500
501     return $line;
502
503 }
```

## 11 Expand Metadata (metadata)

Expand metadata lines into L<sup>A</sup>T<sub>E</sub>X definitions. Maximum of one metadata item per line.

```
511 sub metadata {
512
513     my ($line, $conf) = @_;
```

```

514
515 my $meta = $conf->{meta};
516 my $tag;
517 my $value;

Extract the tag name and value.

520 ($tag, $value) = $line=~/\s*\.[a-zA-Z][a-zA-Z]\s*(.*)/;
521 $tag = lc $tag;
522
523 if ($value) {

    If it's not the date tag ".da" then just append it to any previous values
    of that tag.

526     if ($tag ne "da") {
527         if ($meta->{$tag}) {
528             $meta->{$tag} .= " ";
529         }
530         $meta->{$tag} .= $value;
531     }

    Dates are dealt with specially and parsed into year, month, and day.
    They should always be given in the ISO format (YYYY/MM/DD).

    Since it doesn't make sense to concatenate multiple date tags to-
    gether one will always overwrite any previous value. The date is parsed
    and converted into three special tags (with three letter names so that
    they can't be overwritten by any conventional tags) for the year, month,
    and day for the template to format as it wishes.
    TODO: Support input date formats other than ISO.

543     else {
544         my ($year, $month, $day) = $value=~/(d+)\/(d+)\/(d+)/;
545         if ($year && $month && $day) {

            Remove leading zeros from the month and day.

547             $month =~ s/^0+//;
548             $day =~ s/^0+//;

            Set "special" data metadata.

550             $meta->{dyr} = $year;
551             $meta->{dmo} = $month;
552             $meta->{dda} = $day;
553         }
554     }
555 }
556
557 }

```

## 12 Dump Buffer (dumpbuffer)

Format and dump a buffer to the @formatted buffer.

```

564 sub dumpbuffer {
565
566     my ($buffer) = @_;
567     my $indentsize;

    Exit if the buffer pointer is null.

570     unless ($buffer) {
571         return;
572     }

    If this is an indented buffer then retrieve the indent size.

575     if ($buffer == \@indent) {
576         $indentsize = shift @$buffer;
577     }

    Remove any blank lines from the beginning and end of the buffer, updating
    the code start line pointer.

581     $codestart += tidybuffer($buffer);

    If there's anything left in the buffer then format and dump it to the for-
    matted buffer.

585     if (scalar(@$buffer) > 0) {
586         if ($buffer == \@code) {
587             dumpcode();
588         }
589         elsif ($buffer == \@crossedout) {
590             dumpcrossedout();
591         }
592         elsif ($buffer == \@noindent) {
593             dumpnoindent();
594         }
595         else {
596             dumpindent($indentsize);
597         }
598     }

    Clear the buffer.

601     @$buffer = ();
602
603 }

```

### 13 Dump the Code Buffer (dumpcode)

```

609 sub dumpcode {
610
611     my $line;

    Output code in a lstlisting environment.

614     push @formatted, '\begin{lstlisting}[firstnumber='.$codestart.\n";
615     foreach $line (@code) {

```

Temporary hack so that `\end{lstlisting}` can appear in a listing (I need some slides that use listings working by tomorrow!).

**FIXME:** *Do this properly!*

```

619     $line =~ s/\end{lstlisting}/\end {lstlisting}/g;
620     push @formatted, "$line\n";
621 }
622 push @formatted, '\end {lstlisting}'."\n";
623
624 }
```

## 14 Dump the Crossed-Out Buffer (dumpcrossedout)

```

630 sub dumpcrossedout {
631
632     my $line;

    Crossed-out lines are put in a non-number lstlisting environment with
    the sequence "XXXX" at the beginning of each line. This is set to be converted
    to a \crossedout sequence which the template should define.

637     push @formatted, '\lstset{delim=[il][\crossedout]XXXX}'."\n";
638     push @formatted, '\begin{lstlisting}[numbers=none]'."\n";
639     foreach $line (@crossedout) {

        The XXXX goes just before the text on the line. If the line is blank
        then no XXXX is added.

642         $line =~ s/^(s*)(\S)/$1XXXX$2/;
643         push @formatted, "$line\n";
644     }
645     push @formatted, '\end'.'{lstlisting}'."\n";
646     push @formatted, '\lstset{deletedelim=[il]XXXX}'."\n";
647
648 }
```

## 15 Dump Non-Indented Text (dumpnoindent)

```

654 sub dumpnoindent {
655
656     my $line;

    Allow the vertical bar character to be used to mark verbatim text in the
    commentary.

660     push @formatted, '\MakeShortVerb{|}'."\n";
661     foreach $line (@noindent) {
662         push @formatted, "$line\n";
663     }
664     push @formatted, '\DeleteShortVerb{|}'."\n";
665
666 }
```

## 16 Dump Indented Text (dumpindent)

```
672 sub dumpindent {
673
674     my ($indentsize) = @_ ;
675     my $line;

    Output the text in an indented environment. This should be defined in
    the template. The vertical bar character can be used for verbatim text.

679     push @formatted, '\MakeShortVerb{|}' . "\n";
680     push @formatted, '\begin{indented}' . "$indentsize\n";
681     foreach $line (@indent) {
682         push @formatted, "$line\n";
683     }
684     push @formatted, '\end{indented}' . "\n";
685     push @formatted, '\DeleteShortVerb{|}' . "\n";
686
687 }
```

## 17 Trim Blank Leading & Trailing Lines (tidybuffer)

Remove blank lines from the beginning and end of an array of lines. Takes a reference to the array for speed. Returns the number of lines that were removed from the beginning of the array so that line number counts can be adjusted.

```
697 sub tidybuffer {
698
699     my ($buf) = @_ ;
700     my $removed = 0;

    Remove blank lines from the end of the buffer. (Note that $code[-1] is the
    last line of the buffer.)

704     while (scalar(@$buf)>0 && $buf->[-1]=~/^\s*$/) {
705         pop @$buf;
706     }

    And from the beginning, updating the start line number too.

708     while (scalar(@$buf)>0 && $buf->[0]=~/^\s*$/) {
709         shift @$buf;
710         $removed++;
711     }
712
713     return $removed;
714
715 }
```

## 18 Output the Formatted Buffer in a Template (outputtemplate)

```

721 sub outputtemplate {
722     my ($conf, $outfile) = @_;
723     my $line;
724
725

```

## 18.1 Construct the Template File Name

The template directory is ignored if the template file name starts with a slash. “\$HOME” will get expanded to the value of that environment variable in the template directory.

```

731     $conf->{tempdir} =~ s/\$HOME\b/\$HOME/;
732     my $templatefile;
733     if ($conf->{template} =~ /\^\/) {
734         $templatefile = $conf->{template};
735     }
736     else {
737         $templatefile = "$conf->{tempdir}/$conf->{template}";
738     }

```

Open the template file.

```

741     unless (open TEMPLATE, $templatefile) {
742         die "$myname: Could not open template \"$templatefile\".\n";
743     }

```

## 18.2 Redirect the Output

If a non-blank filename is supplied then redirect all output to that.

```

747     if ($outfile) {
748         unless (open(OUT, ">$outfile")) {
749             die "$myname: Could not open output file \"$outfile\".\n";
750         }
751         select OUT;
752     }

```

## 18.3 Scan the Template

The only valid tags in the template are .me (metadata) and .bd (body). The metadata tag must appear before the body or it'll be ignored.

```

758     while (defined($line = <TEMPLATE>) && $line !~ /\^s*\bd/) {
759         if ($line =~ /\^s*\me/) {
760             Extract list of required metadata.
761             $line =~ s/\^s*\me\s*//;
762             $line = lc $line;
763             $line =~ s/\s*//g;
764             And format it.
765             outputmetadata($conf, split /,/, $line);
766         }
767         else {

```

```

768         print $line;
769     }
770 }

```

We've now either reached the body tag or the end of the template. In either case it's time to dump the formatted output.

```

774 print @formatted;

If there's any more of the template left then print it.

777 while (defined($line = <TEMPLATE>)) {
778     print $line;
779 }
780
781 close TEMPLATE;
782 select STDOUT;
783 close OUT;
784
785 }

```

## 19 Output Metadata (outputmetadata)

Convert the metadata to L<sup>A</sup>T<sub>E</sub>X tags and output it. Takes the configuration hash and a list of meta data types required.

```

793 sub outputmetadata {
794
795     my ($conf, @tags) = @_;
796     my $meta = $conf->{meta};
797     my $tag;
798
799     foreach $tag (@tags) {

        Undefined tags that the template requests are output blank. The tem-
        plate should use something like the ifthen package to detect these and
        supply sensible defaults.

        We used to just not output a value for undefined tags but that
        would cause problems where the output of multiple runs is concatenated
        together since undefined tags in one source file would inherit values from
        source files preceding them.

808         unless (defined $meta->{$tag}) {
809             $meta->{$tag} = "";
810         }
811         print "\\def\\meta$tag\\{$meta->{$tag}\\}\n";
812     }
813
814 }

```

## 20 Quote L<sup>A</sup>T<sub>E</sub>X Special Characters (quotetex)

Used to generate the `fi` item of metadata from the file name.

```

821 sub quotetex {
822
823     my ($text) = @_;
        Escape the backslashes first or you end up escaping your escape codes!
826     $text =~ s/\\\/\textbackslash/g;
        Escape characters that just need a backslash inserting in front of them.
829     $text =~ s/([\$\&#\_{}])\/\textbackslash/g;
        Wrap the \textbackslash we created above in braces to stop problems
        with the characters following it.
833     $text =~ s/(\textbackslash)/{\$1}/g;
        Deal with the last two special characters.
836     $text =~ s/~/{\textasciitilde}/g;
837     $text =~ s/\^{}/{\textasciicircum}/g;
838
839     return $text;
840
841 }

```