```
#!/usr/bin/perl

#
# Markdown -- A text-to-HTML conversion tool for web writers
#
# Copyright (c) 2004 John Gruber
# <http://daringfireball.net/projects/markdown/>
#


package Markdown;
require 5.006_000;
use strict;
use warnings;

use Digest::MD5 qw(md5_hex);
use vars qw($VERSION);
$VERSION = '1.0.1';
# Tue 14 Dec 2004

## Disabled; causes problems under Perl 5.6.1:
# use utf8;
# binmode( STDOUT, ":utf8" );  # c.f.:
http://acis.openlib.org/dev/perl-unicode-struggle.html


#
# Global default settings:
#
my $g_empty_element_suffix = " />";     # Change to ">" for
HTML output
my $g_tab_width = 4;


#
# Globals:
#

# Regex to match balanced [brackets]. See Friedl's
# "Mastering Regular Expressions", 2nd Ed., pp. 328-331.
my $g_nested_brackets;
$g_nested_brackets = qr{
      (?>                                           # Atomic
matching
         [^\[\]]+                                           #
Anything other than brackets
       |
         \[
             (??{ $g_nested_brackets })      # Recursive set of
nested brackets
         \]
      )*
}x;


# Table of hash values for escaped characters:
my %g_escape_table;
```

```
foreach my $char (split //, '\\`*_{}[]()>#+-.!') {
     $g_escape_table{$char} = md5_hex($char);
}


# Global hashes, used by various utility routines
my %g_urls;
my %g_titles;
my %g_html_blocks;

# Used to track when we're inside an ordered or unordered list
# (see _ProcessListItems() for details):
my $g_list_level = 0;


#### Blosxom plug-in interface
##########################################

# Set $g_blosxom_use_meta to 1 to use Blosxom's meta plug-in
to determine
# which posts Markdown should process, using a "meta-markup:
markdown"
# header. If it's set to 0 (the default), Markdown will
process all
# entries.
my $g_blosxom_use_meta = 0;

sub start { 1; }
sub story {
     my($pkg, $path, $filename, $story_ref, $title_ref,
$body_ref) = @_;

     if ( (! $g_blosxom_use_meta) or
          (defined($meta::markup) and ($meta::markup =~ /^
\s*markdown\s*$/i))
          ){
               $$body_ref  = Markdown($$body_ref);
     }
     1;
}


#### Movable Type plug-in interface
#####################################
eval {require MT};  # Test to see if we're running in MT.
unless ($@) {
    require MT;
    import  MT;
    require MT::Template::Context;
    import  MT::Template::Context;

     eval {require MT::Plugin};  # Test to see if we're
running >= MT 3.0.
     unless ($@) {
          require MT::Plugin;
          import  MT::Plugin;
          my $plugin = new MT::Plugin({
```

```
                name => "Markdown",
                description => "A plain-text-to-HTML
formatting plugin. (Version: $VERSION)",
                doc_link =>
'http://daringfireball.net/projects/markdown/'
           });
           MT->add_plugin( $plugin );
     }

     MT::Template::Context->add_container_tag(MarkdownOptions
=> sub {
           my $ctx     = shift;
           my $args = shift;
           my $builder = $ctx->stash('builder');
           my $tokens = $ctx->stash('tokens');

           if (defined ($args->{'output'}) ) {
                $ctx->stash('markdown_output', lc $args->
{'output'});
           }

           defined (my $str = $builder->build($ctx, $tokens) )
                or return $ctx->error($builder->errstr);
           $str;      # return value
     });

     MT->add_text_filter('markdown' => {
           label     => 'Markdown',
           docs      =>
'http://daringfireball.net/projects/markdown/',
           on_format => sub {
                my $text = shift;
                my $ctx  = shift;
                my $raw  = 0;
               if (defined $ctx) {
                my $output = $ctx->stash('markdown_output');
                     if (defined $output  &&  $output =~ m/
^html/i) {
                          $g_empty_element_suffix = ">";
                          $ctx->stash('markdown_output', '');
                     }
                     elsif (defined $output  &&  $output eq
'raw') {
                          $raw = 1;
                          $ctx->stash('markdown_output', '');
                     }
                     else {
                          $raw = 0;
                          $g_empty_element_suffix = " />";
                     }
                }
                $text = $raw ? $text : Markdown($text);
                $text;
           },
     });

     # If SmartyPants is loaded, add a combo
```

```
Markdown/SmartyPants text filter:
      my $smartypants;

      {
            no warnings "once";
            $smartypants =
$MT::Template::Context::Global_filters{'smarty_pants'};
      }

      if ($smartypants) {
            MT->add_text_filter('markdown_with_smartypants' => {
                  label     => 'Markdown With SmartyPants',
                  docs      =>
'http://daringfireball.net/projects/markdown/',
                  on_format => sub {
                        my $text = shift;
                        my $ctx  = shift;
                        if (defined $ctx) {
                              my $output = $ctx->
stash('markdown_output');
                              if (defined $output  &&  $output eq
'html') {
                                    $g_empty_element_suffix = ">";
                              }
                              else {
                                    $g_empty_element_suffix = "
/>";
                              }
                        }
                        $text = Markdown($text);
                        $text = $smartypants->($text, '1');
                  },
            });
      }
}
else {
#### BBEdit/command-line text filter interface
#############################
# Needs to be hidden from MT (and Blosxom when running in
static mode).

    # We're only using $blosxom::version once; tell Perl not
to warn us:
      no warnings 'once';
    unless ( defined($blosxom::version) ) {
            use warnings;

            #### Check for command-line switches:
##################
            my %cli_opts;
            use Getopt::Long;
            Getopt::Long::Configure('pass_through');
            GetOptions(\%cli_opts,
                  'version',
                  'shortversion',
                  'html4tags',
            );
```

```
          if ($cli_opts{'version'}) {      # Version info
               print "\nThis is Markdown, version $VERSION.
\n";
               print "Copyright 2004 John Gruber\n";
               print
"http://daringfireball.net/projects/markdown/\n\n";
               exit 0;
          }
          if ($cli_opts{'shortversion'}) {      # Just the
version number string.
               print $VERSION;
               exit 0;
          }
          if ($cli_opts{'html4tags'}) {               # Use
HTML tag style instead of XHTML
               $g_empty_element_suffix = ">";
          }


          #### Process incoming text:
############################
          my $text;
          {
               local $/;               # Slurp the whole file
               $text = <>;
          }
        print Markdown($text);
    }
}



sub Markdown {
#
# Main function. The order in which other subs are called here
is
# essential. Link and image substitutions need to happen
before
# _EscapeSpecialChars(), so that any *'s or _'s in the <a>
# and <img> tags get encoded.
#
     my $text = shift;

     # Clear the global hashes. If we don't clear these, you
get conflicts
     # from other articles when generating a page which
contains more than
     # one article (e.g. an index page that shows the N most
recent
     # articles):
     %g_urls = ();
     %g_titles = ();
     %g_html_blocks = ();


     # Standardize line endings:
     $text =~ s{\r\n}{\n}g;     # DOS to Unix
```

```
      $text =~ s{\r}{\n}g;  # Mac to Unix

      # Make sure $text ends with a couple of newlines:
      $text .= "\n\n";

      # Convert all tabs to spaces.
      $text = _Detab($text);

      # Strip any lines consisting only of spaces and tabs.
      # This makes subsequent regexen easier to write, because
we can
      # match consecutive blank lines with /\n+/ instead of
something
      # contorted like /[ \t]*\n+/ .
      $text =~ s/^[ \t]+$//mg;

      # Turn block-level HTML blocks into hash entries
      $text = _HashHTMLBlocks($text);

      # Strip link definitions, store in hashes.
      $text = _StripLinkDefinitions($text);

      $text = _RunBlockGamut($text);

      $text = _UnescapeSpecialChars($text);

      return $text . "\n";
}


sub _StripLinkDefinitions {
#
# Strips link definitions from text, stores the URLs and
titles in
# hash references.
#
      my $text = shift;
      my $less_than_tab = $g_tab_width - 1;

      # Link defs are in the form: ^[id]: url "optional title"
      while ($text =~ s{
                                    ^[ ]{0,$less_than_tab}\[(.
+)\]: # id = $1
                                      [ \t]*
                                      \n?                 # maybe
*one* newline
                                      [ \t]*
                                    <?(\S+?)>?            # url =
$2
                                      [ \t]*
                                      \n?                 # maybe
one newline
                                      [ \t]*
                                    (?:
                                          (?<=\s)               #
lookbehind for whitespace
                                          ["(]
```

```
                                        (.+?)           # title
= $3
                                        ["")]
                                        [ \t]*
                                    )?  # title is optional
                                    (?:\n+|\Z)
                                }
                                {}mx) {
            $g_urls{lc $1} = _EncodeAmpsAndAngles( $2 );     #
Link IDs are case-insensitive
            if ($3) {
                  $g_titles{lc $1} = $3;
                  $g_titles{lc $1} =~ s/"/&quot;/g;
            }
      }

      return $text;
}


sub _HashHTMLBlocks {
      my $text = shift;
      my $less_than_tab = $g_tab_width - 1;

      # Hashify HTML blocks:
      # We only want to do this for block-level HTML tags, such
as headers,
      # lists, and tables. That's because we still want to wrap
<p>s around
      # "paragraphs" that are wrapped in non-block-level tags,
such as anchors,
      # phrase emphasis, and spans. The list of tags we're
looking for is
      # hard-coded:
      my $block_tags_a =
qr/p|div|h[1-6]|blockquote|pre|table|dl|ol|ul|script|noscript|
form|fieldset|iframe|math|ins|del/;
      my $block_tags_b =
qr/p|div|h[1-6]|blockquote|pre|table|dl|ol|ul|script|noscript|
form|fieldset|iframe|math/;

      # First, look for nested blocks, e.g.:
      #     <div>
      #           <div>
      #           tags for inner block must be indented.
      #           </div>
      #     </div>
      #
      # The outermost tags must start at the left margin for
this to match, and
      # the inner nested divs must be indented.
      # We need to do this before the next, more liberal match,
because the next
      # match will start at the first `<div>` and stop at the
first `</div>`.
      $text =~ s{
                        (                               # save
```

```
in $1
                        ^                           # start
of line  (with /m)
                        <($block_tags_a) # start tag = $2
                        \b                          # word
break
                        (.*\n)*?            # any number
of lines, minimally matching
                        </\2>               # the
matching end tag
                        [ \t]*                      #
trailing spaces/tabs
                        (?=\n+|\Z) # followed by a newline
or end of document
                    )
                }{
                    my $key = md5_hex($1);
                    $g_html_blocks{$key} = $1;
                    "\n\n" . $key . "\n\n";
                }egmx;


    #
    # Now match more liberally, simply from `\n<tag>` to
`</tag>\n`
    #
    $text =~ s{
                    (                               # save
in $1
                        ^                           # start
of line  (with /m)
                        <($block_tags_b) # start tag = $2
                        \b                          # word
break
                        (.*\n)*?            # any number
of lines, minimally matching
                        .*</\2>                     # the
matching end tag
                        [ \t]*                      #
trailing spaces/tabs
                        (?=\n+|\Z) # followed by a newline
or end of document
                    )
                }{
                    my $key = md5_hex($1);
                    $g_html_blocks{$key} = $1;
                    "\n\n" . $key . "\n\n";
                }egmx;
    # Special case just for <hr />. It was easier to make a
special case than
    # to make the other regex more complicated.
    $text =~ s{
                    (?:
                        (?<=\n\n)       # Starting after a
blank line
                        |                   # or
                        \A\n?           # the beginning of
```

```
the doc
                    )
                    (                               # save
in $1
                         [ ]{0,$less_than_tab}
                         <(hr)                      # start tag =
$2
                         \b                         # word
break
                         ([^<>])*?                  #
                         /?>                        # the
matching end tag
                         [ \t]*
                         (?=\n{2,}|\Z)              # followed by
a blank line or end of document
                    )
               }{
                    my $key = md5_hex($1);
                    $g_html_blocks{$key} = $1;
                    "\n\n" . $key . "\n\n";
               }egx;

     # Special case for standalone HTML comments:
     $text =~ s{
                    (?:
                         (?<=\n\n)        # Starting after a
blank line
                         |                # or
                         \A\n?            # the beginning of
the doc
                    )
                    (                               # save
in $1
                         [ ]{0,$less_than_tab}
                         (?s:
                              <!
                              (--.*?--\s*)+
                              >
                         )
                         [ \t]*
                         (?=\n{2,}|\Z)              # followed by
a blank line or end of document
                    )
               }{
                    my $key = md5_hex($1);
                    $g_html_blocks{$key} = $1;
                    "\n\n" . $key . "\n\n";
               }egx;


     return $text;
}


sub _RunBlockGamut {
#
# These are all the transformations that form block-level
```

```
# tags like paragraphs, headers, and list items.
#
      my $text = shift;

      $text = _DoHeaders($text);

      # Do Horizontal Rules:
      $text =~ s{^[ ]{0,2}([ ]?\*[ ]?){3,}[ \t]*$}{\n<hr
$g_empty_element_suffix\n}gmx;
      $text =~ s{^[ ]{0,2}([ ]? -[ ]?){3,}[ \t]*$}{\n<hr
$g_empty_element_suffix\n}gmx;
      $text =~ s{^[ ]{0,2}([ ]? _[ ]?){3,}[ \t]*$}{\n<hr
$g_empty_element_suffix\n}gmx;

      $text = _DoLists($text);

      $text = _DoCodeBlocks($text);

      $text = _DoBlockQuotes($text);

      # We already ran _HashHTMLBlocks() before, in Markdown(),
but that
      # was to escape raw HTML in the original Markdown source.
This time,
      # we're escaping the markup we've just created, so that
we don't wrap
      # <p> tags around block-level tags.
      $text = _HashHTMLBlocks($text);

      $text = _FormParagraphs($text);

      return $text;
}


sub _RunSpanGamut {
#
# These are all the transformations that occur *within* block-
level
# tags like paragraphs, headers, and list items.
#
      my $text = shift;

      $text = _DoCodeSpans($text);

      $text = _EscapeSpecialChars($text);

      # Process anchor and image tags. Images must come first,
      # because ![foo][f] looks like an anchor.
      $text = _DoImages($text);
      $text = _DoAnchors($text);

      # Make links out of things like `<http://example.com/>`
      # Must come after _DoAnchors(), because you can use
< and >
      # delimiters in inline links like [this](<url>).
      $text = _DoAutoLinks($text);
```

```
      $text = _EncodeAmpsAndAngles($text);

      $text = _DoItalicsAndBold($text);

      # Do hard breaks:
      $text =~ s/ {2,}\n/ <br$g_empty_element_suffix\n/g;

      return $text;
}


sub _EscapeSpecialChars {
      my $text = shift;
      my $tokens ||= _TokenizeHTML($text);

      $text = '';   # rebuild $text from the tokens
#     my $in_pre = 0;   # Keep track of when we're inside <pre>
or <code> tags.
#     my $tags_to_skip = qr!<(/?)(?:pre|code|kbd|script|math)
[\s>]!;

      foreach my $cur_token (@$tokens) {
            if ($cur_token->[0] eq "tag") {
                  # Within tags, encode * and _ so they don't
conflict
                  # with their use in Markdown for italics and
strong.
                  # We're replacing each such character with its
                  # corresponding MD5 checksum value; this is
likely
                  # overkill, but it should prevent us from
colliding
                  # with the escape values by accident.
                  $cur_token->[1] =~  s! \* !
$g_escape_table{'*'}!gx;
                  $cur_token->[1] =~  s! _  !
$g_escape_table{'_'}!gx;
                  $text .= $cur_token->[1];
            } else {
                  my $t = $cur_token->[1];
                  $t = _EncodeBackslashEscapes($t);
                  $text .= $t;
            }
      }
      return $text;
}


sub _DoAnchors {
#
# Turn Markdown link shortcuts into XHTML <a> tags.
#
      my $text = shift;

      #
      # First, handle reference-style links: [link text] [id]
```

```
      #
      $text =~ s{
            (                           # wrap whole match in $1
              \[
                ($g_nested_brackets)    # link text = $2
              \]

              [ ]?                      # one optional space
              (?:\n[ ]*)?         # one optional newline
followed by spaces

              \[
                (.*?)       # id = $3
              \]
            )
      }{
            my $result;
            my $whole_match = $1;
            my $link_text   = $2;
            my $link_id     = lc $3;

            if ($link_id eq "") {
                  $link_id = lc $link_text;     # for shortcut
links like [this][].
            }

            if (defined $g_urls{$link_id}) {
                  my $url = $g_urls{$link_id};
                  $url =~ s! \* !$g_escape_table{'*'}!gx;
      # We've got to encode these to avoid
                  $url =~ s!   _ !$g_escape_table{'_'}!gx;
      # conflicting with italics/bold.
                  $result = "<a href=\"$url\"";
                  if ( defined $g_titles{$link_id} ) {
                        my $title = $g_titles{$link_id};
                        $title =~ s! \* !$g_escape_table{'*'}!gx;
                        $title =~ s!   _ !$g_escape_table{'_'}!gx;
                        $result .=  " title=\"$title\"";
                  }
                  $result .= ">$link_text</a>";
            }
            else {
                  $result = $whole_match;
            }
            $result;
      }xsge;

      #
      # Next, inline-style links: [link text](url "optional
title")
      #
      $text =~ s{
            (                     # wrap whole match in $1
              \[
                ($g_nested_brackets)    # link text = $2
              \]
              \(            # literal paren
```

```
                [ \t]*
                <?(.*?)>?  # href = $3
                [ \t]*
                (                   # $4
                  (['"])    # quote char = $5
                  (.*?)             # Title = $6
                  \5        # matching quote
                )?                  # title is optional
              \)
            )
      }{
            my $result;
            my $whole_match = $1;
            my $link_text   = $2;
            my $url                 = $3;
            my $title       = $6;

            $url =~ s! \* !$g_escape_table{'*'}!gx;           #
We've got to encode these to avoid
            $url =~ s!   _ !$g_escape_table{'_'}!gx;          #
conflicting with italics/bold.
            $result = "<a href=\"$url\"";

            if (defined $title) {
                  $title =~ s/"/&quot;/g;
                  $title =~ s! \* !$g_escape_table{'*'}!gx;
                  $title =~ s!   _ !$g_escape_table{'_'}!gx;
                  $result .=  " title=\"$title\"";
            }

            $result .= ">$link_text</a>";

            $result;
      }xsge;

      return $text;
}


sub _DoImages {
#
# Turn Markdown image shortcuts into <img> tags.
#
      my $text = shift;

      #
      # First, handle reference-style labeled images: ![alt
text][id]
      #
      $text =~ s{
            (                   # wrap whole match in $1
              !\[
                (.*?)       # alt text = $2
              \]

              [ ]?                    # one optional space
              (?:\n[ ]*)?         # one optional newline
```

```
followed by spaces

              \[
                (.*?)       # id = $3
              \]

            )
      }{
            my $result;
            my $whole_match = $1;
            my $alt_text    = $2;
            my $link_id     = lc $3;

            if ($link_id eq "") {
                  $link_id = lc $alt_text;     # for shortcut
links like ![this][].
            }

            $alt_text =~ s/"/&quot;/g;
            if (defined $g_urls{$link_id}) {
                  my $url = $g_urls{$link_id};
                  $url =~ s! \* !$g_escape_table{'*'}!gx;
      # We've got to encode these to avoid
                  $url =~ s!   _ !$g_escape_table{'_'}!gx;
      # conflicting with italics/bold.
                  $result = "<img src=\"$url\" alt=\"$alt_text
\"";
                  if (defined $g_titles{$link_id}) {
                        my $title = $g_titles{$link_id};
                        $title =~ s! \* !$g_escape_table{'*'}!gx;
                        $title =~ s!   _ !$g_escape_table{'_'}!gx;
                        $result .=  " title=\"$title\"";
                  }
                  $result .= $g_empty_element_suffix;
            }
            else {
                  # If there's no such link ID, leave intact:
                  $result = $whole_match;
            }

            $result;
      }xsge;

      #
      # Next, handle inline images:  ![alt text](url "optional
title")
      # Don't forget: encode * and _

      $text =~ s{
            (                     # wrap whole match in $1
              !\[
                (.*?)       # alt text = $2
              \]
              \(            # literal paren
                  [ \t]*
                  <?(\S+?)>? # src url = $3
                  [ \t]*
```

```
                (                       # $4
                  (['"])    # quote char = $5
                  (.*?)             # title = $6
                  \5        # matching quote
                  [ \t]*
                )?                      # title is optional
              \)
            )
      }{
            my $result;
            my $whole_match = $1;
            my $alt_text    = $2;
            my $url                 = $3;
            my $title       = '';
            if (defined($6)) {
                  $title            = $6;
            }

            $alt_text =~ s/"/&quot;/g;
            $title    =~ s/"/&quot;/g;
            $url =~ s! \* !$g_escape_table{'*'}!gx;           #
We've got to encode these to avoid
            $url =~ s!   _ !$g_escape_table{'_'}!gx;          #
conflicting with italics/bold.
            $result = "<img src=\"$url\" alt=\"$alt_text\"";
            if (defined $title) {
                  $title =~ s! \* !$g_escape_table{'*'}!gx;
                  $title =~ s!   _ !$g_escape_table{'_'}!gx;
                  $result .=  " title=\"$title\"";
            }
            $result .= $g_empty_element_suffix;

            $result;
      }xsge;

      return $text;
}


sub _DoHeaders {
      my $text = shift;

      # Setext-style headers:
      #       Header 1
      #       ========
      #
      #       Header 2
      #       --------
      #
      $text =~ s{ ^(.+)[ \t]*\n=+[ \t]*\n+ }{
            "<h1>"  .  _RunSpanGamut($1)  .  "</h1>\n\n";
      }egmx;

      $text =~ s{ ^(.+)[ \t]*\n-+[ \t]*\n+ }{
            "<h2>"  .  _RunSpanGamut($1)  .  "</h2>\n\n";
      }egmx;
```

```
      # atx-style headers:
      #     # Header 1
      #     ## Header 2
      #     ## Header 2 with closing hashes ##
      #     ...
      #     ###### Header 6
      #
      $text =~ s{
                  ^(\#{1,6}) # $1 = string of #'s
                  [ \t]*
                  (.+?)      # $2 = Header text
                  [ \t]*
                  \#*              # optional closing #'s (not
counted)
                  \n+
            }{
                  my $h_level = length($1);
                  "<h$h_level>"  .  _RunSpanGamut($2)  .  "</h
$h_level>\n\n";
            }egmx;

      return $text;
}


sub _DoLists {
#
# Form HTML ordered (numbered) and unordered (bulleted) lists.
#
      my $text = shift;
      my $less_than_tab = $g_tab_width - 1;

      # Re-usable patterns to match list item bullets and
number markers:
      my $marker_ul  = qr/[*+-]/;
      my $marker_ol  = qr/\d+[.]/;
      my $marker_any = qr/(?:$marker_ul|$marker_ol)/;

      # Re-usable pattern to match any entirel ul or ol list:
      my $whole_list = qr{
            (                                               # $1 =
whole list
              (                                             # $2
                  [ ]{0,$less_than_tab}
                  (${marker_any})                  # $3 = first
list item marker
                  [ \t]+
              )
              (?s:.+?)
              (                                             # $4
                    \z
                  |
                    \n{2,}
                    (?=\S)
                    (?!                                 # Negative
lookahead for another list item marker
```

```
                    [ \t]*
                    ${marker_any}[ \t]+
                )
            )
        )
    }mx;

    # We use a different prefix before nested lists than top-
level lists.
    # See extended comment in _ProcessListItems().
    #
    # Note: There's a bit of duplication here. My original
implementation
    # created a scalar regex pattern as the conditional
result of the test on
    # $g_list_level, and then only ran the $text =~ s{...}
{...}egmx
    # substitution once, using the scalar as the pattern.
This worked,
    # everywhere except when running under MT on my hosting
account at Pair
    # Networks. There, this caused all rebuilds to be killed
by the reaper (or
    # perhaps they crashed, but that seems incredibly
unlikely given that the
    # same script on the same server ran fine *except* under
MT. I've spent
    # more time trying to figure out why this is happening
than I'd like to
    # admit. My only guess, backed up by the fact that this
workaround works,
    # is that Perl optimizes the substition when it can
figure out that the
    # pattern will never change, and when this optimization
isn't on, we run
    # afoul of the reaper. Thus, the slightly redundant code
to that uses two
    # static s/// patterns rather than one conditional
pattern.

    if ($g_list_level) {
        $text =~ s{
                    ^
                    $whole_list
                }{
                    my $list = $1;
                    my $list_type = ($3 =~ m/$marker_ul/) ?
"ul" : "ol";
                    # Turn double returns into triple
returns, so that we can make a
                    # paragraph for the last item in a list,
if necessary:
                    $list =~ s/\n{2,}/\n\n\n/g;
                    my $result = _ProcessListItems($list,
$marker_any);
                    $result = "<$list_type>\n" . $result .
"</$list_type>\n";
```

```
                        $result;
                }egmx;
        }
        else {
                $text =~ s{
                        (?:(?<=\n\n)|\A\n?)
                        $whole_list
                }{
                        my $list = $1;
                        my $list_type = ($3 =~ m/$marker_ul/) ?
"ul" : "ol";
                        # Turn double returns into triple
returns, so that we can make a
                        # paragraph for the last item in a list,
if necessary:
                        $list =~ s/\n{2,}/\n\n\n/g;
                        my $result = _ProcessListItems($list,
$marker_any);
                        $result = "<$list_type>\n" . $result .
"</$list_type>\n";
                        $result;
                }egmx;
        }


        return $text;
}


sub _ProcessListItems {
#
#       Process the contents of a single ordered or unordered
list, splitting it
#       into individual list items.
#

        my $list_str = shift;
        my $marker_any = shift;


        # The $g_list_level global keeps track of when we're
inside a list.
        # Each time we enter a list, we increment it; when we
leave a list,
        # we decrement. If it's zero, we're not in a list
anymore.
        #
        # We do this because when we're not inside a list, we
want to treat
        # something like this:
        #
        #           I recommend upgrading to version
        #           8. Oops, now this line is treated
        #           as a sub-list.
        #
        # As a single paragraph, despite the fact that the second
line starts
```

```
      # with a digit-period-space sequence.
      #
      # Whereas when we're inside a list (or sub-list), that
line will be
      # treated as the start of a sub-list. What a kludge, huh?
This is
      # an aspect of Markdown's syntax that's hard to parse
perfectly
      # without resorting to mind-reading. Perhaps the solution
is to
      # change the syntax rules such that sub-lists must start
with a
      # starting cardinal number; e.g. "1." or "a.".

      $g_list_level++;

      # trim trailing blank lines:
      $list_str =~ s/\n{2,}\z/\n/;


      $list_str =~ s{
            (\n)?                                   # leading
line = $1
            (^[ \t]*)                               # leading
whitespace = $2
            ($marker_any) [ \t]+              # list marker = $3
            ((?s:.+?)                               # list item
text   = $4
            (\n{1,2}))
            (?= \n* (\z | \2 ($marker_any) [ \t]+))
      }{
            my $item = $4;
            my $leading_line = $1;
            my $leading_space = $2;

            if ($leading_line or ($item =~ m/\n{2,}/)) {
                  $item = _RunBlockGamut(_Outdent($item));
            }
            else {
                  # Recursion for sub-lists:
                  $item = _DoLists(_Outdent($item));
                  chomp $item;
                  $item = _RunSpanGamut($item);
            }

            "<li>" . $item . "</li>\n";
      }egmx;

      $g_list_level--;
      return $list_str;
}



sub _DoCodeBlocks {
#
#     Process Markdown `<pre><code>` blocks.
```

```
#

      my $text = shift;

      $text =~ s{
                  (?:\n\n|\A)
                  (                # $1 = the code block -- one
or more lines, starting with a space/tab
                    (?:
                      (?:[ ]{$g_tab_width} | \t)  # Lines must
start with a tab or a tab-width of spaces
                      .*\n+
                    )+
                  )
                  ((?=^[ ]{0,$g_tab_width}\S)|\Z)  # Lookahead
for non-space at line-start, or end of doc
            }{
                  my $codeblock = $1;
                  my $result; # return value

                  $codeblock =
_EncodeCode(_Outdent($codeblock));
                  $codeblock = _Detab($codeblock);
                  $codeblock =~ s/\A\n+//; # trim leading
newlines
                  $codeblock =~ s/\s+\z//; # trim trailing
whitespace

                  $result = "\n\n<pre><code>" . $codeblock .
"\n</code></pre>\n\n";

                  $result;
            }egmx;

      return $text;
}


sub _DoCodeSpans {
#
#     *     Backtick quotes are used for <code></code> spans.
#
#     *     You can use multiple backticks as the delimiters if
you want to
#           include literal backticks in the code span. So, this
input:
#
#         Just type ``foo `bar` baz`` at the prompt.
#
#           Will translate to:
#
#         <p>Just type <code>foo `bar` baz</code> at the
prompt.</p>
#
#           There's no arbitrary limit to the number of
backticks you
#           can use as delimters. If you need three consecutive
```

```
backticks
#          in your code, use four for delimiters, etc.
#
#     *    You can use spaces to get literal backticks at the
edges:
#
#          ... type `` `bar` `` ...
#
#          Turns to:
#
#          ... type <code>`bar`</code> ...
#

      my $text = shift;

      $text =~ s@
                  (`+)        # $1 = Opening run of `
                  (.+?)       # $2 = The code block
                  (?<!`)
                  \1                # Matching closer
                  (?!`)
            @
                  my $c = "$2";
                  $c =~ s/^[ \t]*//g; # leading whitespace
                  $c =~ s/[ \t]*$//g; # trailing whitespace
                  $c = _EncodeCode($c);
                  "<code>$c</code>";
            @egsx;

      return $text;
}


sub _EncodeCode {
#
# Encode/escape certain characters inside Markdown code runs.
# The point is that in code, these characters are literals,
# and lose their special Markdown meanings.
#
    local $_ = shift;

      # Encode all ampersands; HTML entities are not
      # entities within a Markdown code span.
      s/&/&amp;/g;

      # Encode $'s, but only if we're running under Blosxom.
      # (Blosxom interpolates Perl variables in article
bodies.)
      {
            no warnings 'once';
      if (defined($blosxom::version)) {
            s/\$/&#036;/g;
      }
    }


      # Do the angle bracket song and dance:
```

```
      s! <  !&lt;!gx;
      s! >  !&gt;!gx;

      # Now, escape characters that are magic in Markdown:
      s! \* !$g_escape_table{'*'}!gx;
      s! _  !$g_escape_table{'_'}!gx;
      s! {  !$g_escape_table{'{'}!gx;
      s! }  !$g_escape_table{'}'}!gx;
      s! \[ !$g_escape_table{'['}!gx;
      s! \] !$g_escape_table{']'}!gx;
      s! \\ !$g_escape_table{'\\'}!gx;

      return $_;
}


sub _DoItalicsAndBold {
      my $text = shift;

      # <strong> must go first:
      $text =~ s{ (\*\*|__) (?=\S) (.+?[*_]*) (?<=\S) \1 }
            {<strong>$2</strong>}gsx;

      $text =~ s{ (\*|_) (?=\S) (.+?) (?<=\S) \1 }
            {<em>$2</em>}gsx;

      return $text;
}


sub _DoBlockQuotes {
      my $text = shift;

      $text =~ s{
              (                                           # Wrap
whole match in $1
                  (
                    ^[ \t]*>[ \t]?           # '>' at the start
of a line
                      .+\n                          # rest of the
first line
                    (.+\n)*                         # subsequent
consecutive lines
                    \n*                             # blanks
                  )+
              )
            }{
                  my $bq = $1;
                  $bq =~ s/^[ \t]*>[ \t]?//gm;      # trim one
level of quoting
                  $bq =~ s/^[ \t]+$//mg;                # trim
whitespace-only lines
                  $bq = _RunBlockGamut($bq);        # recurse

                  $bq =~ s/^/  /g;
                  # These leading spaces screw with <pre>
content, so we need to fix that:
```

```
                $bq =~ s{
                            (\s*<pre>.+?</pre>)
                      }{
                            my $pre = $1;
                            $pre =~ s/^  //mg;
                            $pre;
                      }egsx;

                "<blockquote>\n$bq\n</blockquote>\n\n";
          }egmx;


      return $text;
}


sub _FormParagraphs {
#
#     Params:
#           $text - string to process with html <p> tags
#
      my $text = shift;

      # Strip leading and trailing lines:
      $text =~ s/\A\n+//;
      $text =~ s/\n+\z//;

      my @grafs = split(/\n{2,}/, $text);

      #
      # Wrap <p> tags.
      #
      foreach (@grafs) {
            unless (defined( $g_html_blocks{$_} )) {
                  $_ = _RunSpanGamut($_);
                  s/^([ \t]*)/<p>/;
                  $_ .= "</p>";
            }
      }

      #
      # Unhashify HTML blocks
      #
      foreach (@grafs) {
            if (defined( $g_html_blocks{$_} )) {
                  $_ = $g_html_blocks{$_};
            }
      }

      return join "\n\n", @grafs;
}


sub _EncodeAmpsAndAngles {
# Smart processing for ampersands and angle brackets that need
to be encoded.
```

```
      my $text = shift;

      # Ampersand-encoding based entirely on Nat Irons's
Amputator MT plugin:
      #   http://bumppo.net/projects/amputator/
      $text =~ s/&(?!#?[xX]?(?:[0-9a-fA-F]+|\w+);)/&amp;/g;

      # Encode naked <'s
      $text =~ s{<(?![a-z/?\$!])}{&lt;}gi;

      return $text;
}


sub _EncodeBackslashEscapes {
#
#   Parameter:  String.
#   Returns:    The string, with after processing the
following backslash
#               escape sequences.
#
    local $_ = shift;

    s! \\\\  !$g_escape_table{'\\'}!gx;          # Must
process escaped backslashes first.
    s! \\`   !$g_escape_table{'`'}!gx;
    s! \\*   !$g_escape_table{'*'}!gx;
    s! \\_   !$g_escape_table{'_'}!gx;
    s! \\{   !$g_escape_table{'{'}!gx;
    s! \\}   !$g_escape_table{'}'}!gx;
    s! \\[   !$g_escape_table{'['}!gx;
    s! \\]   !$g_escape_table{']'}!gx;
    s! \\(   !$g_escape_table{'('}!gx;
    s! \\)   !$g_escape_table{')'}!gx;
    s! \\>   !$g_escape_table{'>'}!gx;
    s! \\#   !$g_escape_table{'#'}!gx;
    s! \\+   !$g_escape_table{'+'}!gx;
    s! \\-   !$g_escape_table{'-'}!gx;
    s! \\.   !$g_escape_table{'.'}!gx;
    s{ \\!   }{$g_escape_table{'!'}}gx;

    return $_;
}


sub _DoAutoLinks {
      my $text = shift;

      $text =~ s{<((https?|ftp):[^'">\s]+)>}{<a
href="$1">$1</a>}gi;

      # Email addresses: <address@domain.foo>
      $text =~ s{
            <
        (?:mailto:)?
            (
                  [-.\w]+
```

```
                \@
                [-a-z0-9]+(\.[-a-z0-9]+)*\.[a-z]+
            )
            >
        }{
            _EncodeEmailAddress( _UnescapeSpecialChars($1) );
        }egix;

        return $text;
}


sub _EncodeEmailAddress {
#
#     Input: an email address, e.g. "foo@example.com"
#
#     Output: the email address as a mailto link, with each
character
#          of the address encoded as either a decimal or hex
entity, in
#          the hopes of foiling most address harvesting spam
bots. E.g.:
#
#       <a href="&#x6D;&#97;&#105;&#108;&#x74;&#111;:&#102;&#
111;&#111;&#64;&#101;
#        x&#x61;&#109;&#x70;&#108;&#x65;&#x2E;&#99;&#111;&#
109;">&#102;&#111;&#111;
#        &#64;&#101;x&#x61;&#109;&#x70;&#108;&#x65;&#x2E;&#99;
&#111;&#109;</a>
#
#     Based on a filter by Matthew Wickline, posted to the
BBEdit-Talk
#     mailing list: <http://tinyurl.com/yu7ue>
#

        my $addr = shift;

        srand;
        my @encode = (
            sub { '&#' .                 ord(shift)   . ';' },
            sub { '&#x' . sprintf( "%X", ord(shift) ) . ';' },
            sub {                            shift          },
        );

        $addr = "mailto:" . $addr;

        $addr =~ s{(.)}{
            my $char = $1;
            if ( $char eq '@' ) {
                # this *must* be encoded. I insist.
                $char = $encode[int rand 1]->($char);
            } elsif ( $char ne ':' ) {
                # leave ':' alone (to spot mailto: later)
                my $r = rand;
                # roughly 10% raw, 45% hex, 45% dec
                $char = (
                    $r > .9   ?  $encode[2]->($char)  :
```

```
                    $r < .45  ?  $encode[1]->($char)  :
                                      $encode[0]->($char)
                );
            }
            $char;
      }gex;

      $addr = qq{<a href="$addr">$addr</a>};
      $addr =~ s{">.+?:}{">}; # strip the mailto: from the
visible part

      return $addr;
}


sub _UnescapeSpecialChars {
#
# Swap back in all the special characters we've hidden.
#
      my $text = shift;

      while( my($char, $hash) = each(%g_escape_table) ) {
            $text =~ s/$hash/$char/g;
      }
    return $text;
}


sub _TokenizeHTML {
#
#   Parameter:  String containing HTML markup.
#   Returns:    Reference to an array of the tokens comprising
the input
#               string. Each token is either a tag (possibly
with nested,
#               tags contained therein, such as <a
href="<MTFoo>">, or a
#               run of text between tags. Each element of the
array is a
#               two-element array; the first is either 'tag'
or 'text';
#               the second is the actual value.
#
#
#   Derived from the _tokenize() subroutine from Brad Choate's
MTRegex plugin.
#       <http://www.bradchoate.com/past/mtregex.php>
#

    my $str = shift;
    my $pos = 0;
    my $len = length $str;
    my @tokens;

    my $depth = 6;
    my $nested_tags = join('|', ('(?:<[a-z/!$](?:[^<>]') x
$depth) . (')*>)' x  $depth);
```

```
    my $match = qr/(?s: <! ( -- .*? -- \s* )+ > ) |  # comment
                   (?s: <\? .*? \?> ) |              #
processing instruction
                   $nested_tags/ix;                   # nested
tags

    while ($str =~ m/($match)/g) {
        my $whole_tag = $1;
        my $sec_start = pos $str;
        my $tag_start = $sec_start - length $whole_tag;
        if ($pos < $tag_start) {
            push @tokens, ['text', substr($str, $pos,
$tag_start - $pos)];
        }
        push @tokens, ['tag', $whole_tag];
        $pos = pos $str;
    }
    push @tokens, ['text', substr($str, $pos, $len - $pos)] if
$pos < $len;
    \@tokens;
}


sub _Outdent {
#
# Remove one level of line-leading tabs or spaces
#
     my $text = shift;

     $text =~ s/^(\t|[ ]{1,$g_tab_width})//gm;
     return $text;
}


sub _Detab {
#
# Cribbed from a post by Bart Lateur:
# <http://www.nntp.perl.org/group/perl.macperl.anyperl/154>
#
     my $text = shift;

     $text =~ s{(.*?)\t}{$1.(' ' x ($g_tab_width - length($1)
% $g_tab_width))}ge;
     return $text;
}


1;

__END__


=pod

=head1 NAME

B<Markdown>
```

```
=head1 SYNOPSIS

B<Markdown.pl> [ B<--html4tags> ] [ B<--version> ] [ B<-
shortversion> ]
    [ I<file> ... ]


=head1 DESCRIPTION

Markdown is a text-to-HTML filter; it translates an easy-to-
read /
easy-to-write structured text format into HTML. Markdown's
text format
is most similar to that of plain text email, and supports
features such
as headers, *emphasis*, code blocks, blockquotes, and links.

Markdown's syntax is designed not as a generic markup
language, but
specifically to serve as a front-end to (X)HTML. You can  use
span-level
HTML tags anywhere in a Markdown document, and you can use
block level
HTML tags (like <div> and <table> as well).

For more information about Markdown's syntax, see:

    http://daringfireball.net/projects/markdown/


=head1 OPTIONS

Use "--" to end switch parsing. For example, to open a file
named "-z", use:

     Markdown.pl -- -z

=over 4


=item B<--html4tags>

Use HTML 4 style for empty element tags, e.g.:

    <br>

instead of Markdown's default XHTML style tags, e.g.:

    <br />


=item B<-v>, B<--version>

Display Markdown's version number and copyright information.
```

```
=item B<-s>, B<--shortversion>

Display the short-form version number.


=back



=head1 BUGS

To file bug reports or feature requests (other than topics
listed in the
Caveats section above) please send email to:

    support@daringfireball.net

Please include with your report: (1) the example input; (2)
the output
you expected; (3) the output Markdown actually produced.


=head1 VERSION HISTORY

See the readme file for detailed release notes for this
version.

1.0.1 - 14 Dec 2004

1.0 - 28 Aug 2004


=head1 AUTHOR

    John Gruber
    http://daringfireball.net

    PHP port and other contributions by Michel Fortin
    http://michelf.com


=head1 COPYRIGHT AND LICENSE

Copyright (c) 2003-2004 John Gruber
<http://daringfireball.net/>
All rights reserved.

Redistribution and use in source and binary forms, with or
without
modification, are permitted provided that the following
conditions are
met:

* Redistributions of source code must retain the above
copyright notice,
  this list of conditions and the following disclaimer.
```

```
* Redistributions in binary form must reproduce the above
copyright
  notice, this list of conditions and the following disclaimer
in the
  documentation and/or other materials provided with the
distribution.

* Neither the name "Markdown" nor the names of its
contributors may
  be used to endorse or promote products derived from this
software
  without specific prior written permission.

This software is provided by the copyright holders and
contributors "as
is" and any express or implied warranties, including, but not
limited
to, the implied warranties of merchantability and fitness for
a
particular purpose are disclaimed. In no event shall the
copyright owner
or contributors be liable for any direct, indirect,
incidental, special,
exemplary, or consequential damages (including, but not
limited to,
procurement of substitute goods or services; loss of use,
data, or
profits; or business interruption) however caused and on any
theory of
liability, whether in contract, strict liability, or tort
(including
negligence or otherwise) arising in any way out of the use of
this
software, even if advised of the possibility of such damage.

=cut
```