

nofake - simple **notangle** replacement for the **noweb** literate programming tool

January 1, 2022

Contents

1	nofake	1
1.1	Manual Page	2
1.2	Implementation	6
1.3	Variables	7
1.4	Functions	8
2	nofake.bat	19
3	nofake.sh	21
4	List of all chunks	23

1 nofake

nofake is a Perl script that acts as a simple substitute for the **notangle** command from the **noweb** literate programming system. When using **noweb**, program documentation and source code are kept together in a single file (by convention named with the **.nw** suffix). The **notangle** command takes such a *noweb* file and extracts the source code from it for compilation. However, not everybody who wants to compile the source code has a full **noweb** installation available. For these cases **nofake** can be shipped with the source code to provide a substitute for the **notangle** command. **nofake** has a much simpler and less general architecture than **notangle** but can handle most cases correctly.

nofake is derived from NegWeb 1.0.1 that was released by its author Darrell Johnson <darrell@boswa.com> into the public domain at <http://boswa.com/misc/negweb/>.

Note that this source is itself a literate program in **noweb** syntax. It has to use the delimiters << and >> literally and hence has to escape them with a preceding @.

1.1 Manual Page

Perl manual pages are written in Perl's Plain Old Document (POD) format. They can be included into a script and extracted from there: `pod2man nofake > nofake.1`; various other format can be generated as well, including HTML. This makes sure they do not get lost when the script is passed around. For the syntax, see the `perlpod` manual.

2 *<manual page in POD format 2>*≡ (6)
=head1 NAME

`nofake` - notangle replacement for the noweb literate programming tool

=head1 SYNOPSIS

`B<nofake> [B<-R>I<chunk> ...] [B<-L>[I<format>]] [B<--dump> state]
[B<--load> state] [I<file> ...]`

`B<nofake> [B<--version> | B<-v>]`

`B<nofake> [B<--list-all>] [I<file> ...]`

`B<nofake> [B<--list-roots>] [I<file> ...]`

=head1 DESCRIPTION

Noweb(1) is a literate-programming tool like Knuth's WEB. A noweb file contains program source code interleaved with documentation. Extracting the source code for compilation requires `notangle(1)`. To allow source code to be shipped to users not using `noweb(1)`, `B<nofake>` offers the most commonly used functionality of `notangle(1)` as a simple `perl(1)` script. Alas, `B<nofake>` extracts source code from a file in `noweb(1)` syntax: `B<nofake>` reads `I<file>` and extracts the code chunk named `I<chunk>` to `stdout`. If no `I<file>` is provided, `B<nofake>` reads from `stdin`, if no `I<chunk>` is named, `B<nofake>` extracts the chunk `C<*>`.

=head1 OPTIONS

=over 4

=item `B<-R>I<chunk>`

Extract chunk `I<chunk>` (recursively) from the `B<noweb>` file and write it to `stdout`.

=item `B<-L>[I<format>]`

B<nofake> emits cpp(1)-style C<#line> directives to allow a compiler emit error messages that refer to I<file> rather than the extracted source code directly. The optional I<format> allows to provided the format of the line directive: C<-L'#line %L "%F"%N'>. In I<format> C<%F> indicates the name of the source file, C<%L> the line number, and C<%N> a newline. The default C<#line %L "%F"%N> is suitable for C compilers.

=item B<--list-all>

List all I<chunks> in B<noweb> files.

=item B<--list-roots>

List all I<chunks> in B<noweb> files that are not referenced by other chunks.

=item B<--dump> state

Save the state after reading B<noweb> sources.

=item B<--load> state

Load a previously dumped state, this can speed things up if processing a large set of documents and extracting various chunks individually from such set.

The dumping and loading of states shouldn't affect normal nofake operation, all other options are available. Please note however that:

- This is not supported by B<notangle>.
- If loading state from stdin (using - as state), it is not possible to read noweb document from stdin.
- If dumping state to stdout (using - as state), no -R option should be given.
- The default behaviour of reading from stdin is preserved if no input files are given, unless loading state from stdin. Thus, when dumping and loading state, these works as expected:

```
nofake -Rdefaults --dump state nofake.nw
```

```
nofake -Rdefaults --load - <state
```

```
nofake -Rdefaults --load state </dev/null
```

```
cat state | nofake -Rdefaults --load -
```

```
nofake --dump - nofake.nw | nofake -Rdefaults --load -
```

```
nofake --dump - nofake.nw | nofake -Rdefaults --load - nofake.nw
```

```
=back
```

```
=head1 SYNTAX OF NOWEB FILES
```

The authoritative source for the syntax of noweb files is the noweb(1) documentation. However, here is an example:

```
<<hello.c>>=
<<includes>>

int main(int argc, char** argv)
{
    <<say hello>>
    return 0;
}

<<say hello>>=
printf("Hello World!\n");
@

<<includes>>=
#include <stdio.h> /* for printf */
@
```

A chunk is defined by C<E<lt>E<lt>chunkE<gt>E<gt>=> and reaches up to the next definition or a line starting with C<@> followed by a space or newline. A chunk can recursively refer to other chunks: chunk C<hello.c> refers to C<includes> and C<say hello>. A chunk is referred to by C<E<lt>E<lt>chunkE<gt>E<gt>>. To use the C<E<lt>E<lt>> and C<E<gt>E<gt>> character literally in a program, precede them with a C<@>. Double C<@> on the first column to put a literal C<@> there, applies only to the first column.

```
=head1 LIMITATIONS
```

The B<nofake> architecture is simpler than that of notangle(1) and therefore one thing do not work. In particular:

```
=over 4
```

```
=item *
```

B<nofake> does not accept the B<-filter> I<command> option that B<notangle> uses to filter chunks before they are emitted.

=back

=head1 COPYING

This software is in the public domain.

THIS SOFTWARE IS PROVIDED ‘‘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 AUTHOR AND COPYRIGHT HOLDER 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.

=head1 AUTHOR

Christian Lindig <lindig@eecs.harvard.edu>

Please note that this is a derived work and is not maintained by Lindig, the canonical source for this version is <https://github.com/ctarbide/ctweb/blob/master/tools/nofake.nw>.

=head1 SEE ALSO

noweb(1), notangle(1), perl(1), cpp(1)

Norman Ramsey, Literate programming simplified, IEEE Software 11(5):97-105, September 1994.

1.2 Implementation

```

6  <nofake 6>≡
    #!/usr/bin/env perl
    #
    # Generated from nofake.nw. Do not edit! Edit nofake.nw instead and
    # run nofake on it:
    #
    # ./nofake -Rnofake nofake.nw > nofake.pl
    # mv nofake.pl nofake
    # chmod a+x nofake
    #
    # The manual page is at the end of this file in Perl's
    # POD format. You can format it using pod2man(1):
    #
    # pod2man nofake > nofake.1
    # nroff -man nofake.1 | more
    #
    # The noweb source for this version is available at:
    #
    # https://github.com/ctarbide/ctweb/blob/master/tools/nofake.nw
    #
    # This software is in the public domain; for details, see the manual
    # page.

    eval 'exec perl -wS $0 ${1+"$@"}'
        if 0;

    use 5.008; # perl v5.8.0 was released on July 18, 2002
    use strict;
    use warnings FATAL => qw{uninitialized void inplace};
    use Carp ();
    $SIG{__DIE__} = \&Carp::confess;

    local $\ = "\n";

    sub version {
        print <<'EOF';
        (c) 2021 C. Tarbide <ctarbide@tuta.io>
        (c) 2002 Christian Lindig <lindig@eecs.harvard.edu>

        NoFake is derived from the public domain NegWeb 1.0.1
        http://boswa.com/misc/negweb by Darrell Johnson <darrell@boswa.com>.
    EOF
    }

```

<defaults 7>

<many utils 18>

<line directive 8a>

<read file 9b>

<sub extract 13b>

<sub usage 13c>

<process command line and extract chunk 16>

--END--

<manual page in POD format 2>

1.3 Variables

All (code) chunks are stored in the global hash table `chunks`. A chunk may contain references to other chunks. These are ignored when a chunk is read and expanded when then chunk itself is expanded.

```
7  <defaults 7>≡ (6)
    my $lineformat = '#line %L "%F"%N';

    # do not emit #line directives
    my $sync = 0;

    # hash tables for chunks and chunks options
    my %chunks = ();
    my %chunks_options = ();

    my $list_roots = 0;
```

1.4 Functions

If we compile source code that was extracted from a noweb file, we want the error messages point to the noweb file and not the file extracted from it. Therefore we emit `#line` directives that connect the extracted file back with the noweb file. They are understood by many compilers such that they can emit error messages pointing to the noweb file.

The `line_directive` returns the string to be used as a line directive. The formatting is taken from `lineformat` that is controlled by the `-L` command line option.

8a \langle line directive 8a $\rangle \equiv$ (6)

```

sub line_directive {
  my ($fname, $line) = @_ ;
  my $ret = $lineformat;
  $ret =~ s/%F/$fname/g;
  $ret =~ s/%L/$line/g;
  $ret =~ s/%N/\n/g;
  return $ret;
}

```

The `read_line` function reads a file line-by-line and recognizes chunk definitions. Each chunk is put under its name into the global `chunks` hash table. The `sync` flag controls whether line directives are emitted.

Relevant `man noweb` section: Documentation chunks begin with a line that starts with an `@` followed by a space or newline.

A `@@` at the first column must yield a `@`.

8b \langle warn unescaped << in documentation chunk 8b $\rangle \equiv$ (9a)

```

Carp::carp("WARNING: Unescaped << in documentation" .
  " chunk at line ${ARGV}:${.}.");

```

8c \langle read line vars 8c $\rangle \equiv$ (9b)

```

my $chunk = '';

```



```

9a  <read line 9a>≡ (9b)
    if ($line =~ m{^<<(.*?)>>=\\s*$}) {
        $chunk = $1;
        if (!$chunks_options{$chunk}) {
            $chunks_options{$chunk} = {nextractions => 0};
        }
        $chunks_options{$chunk}->{nchunks}++;
    } elsif ($chunk) {
        if ($line =~ m{^\\@(?:$|\\s)}) {
            $chunk = '';
        } else {
            # regular line inside chunk
            $line =~ s,^\\@\\@,\\@,;
            set_many(%chunks, $chunk, $ARGV, $., $line . "\\n");
        }
    } else {
        if ($line =~ m{^(|.*?[\\@])<<} and $1 !~ m{\\[\\[]}) {
            <warn unescaped << in documentation chunk 8b>
        }
    }
}

```

```

9b  <read file 9b>≡ (6)
    sub read_file {
        local @ARGV = @_;
        <read line vars 8c>
        while (my $line = <>) {
            chomp($line);
            <read line 9a>
        }
    }

```

The **extract** function takes a chunk name and extracts this chunk recursively from the **chunks** hash table. The output is returned as a string.

While we look for chunk names in lines to extract we have to be careful: if a chunk name delimiter is preceded by a an **@** it does not denote a chunk, but the delimiter literally.

When dealing with line numbers, **nofake** prioritize document structure, **notangle** is more liberal in breaking lines to put line directives.

```

9c  <setup $before_has_content 9c>≡ (10a 11)
    my $before_has_content = $before !~ m{^\\s*$};

```

10a *<setup \$indent and \$before_has_content 10a>*≡ (11)

```

my $indent;
<setup $before_has_content 9c>
if ($before_has_content) {
    $indent = ' ' x length($before);
} else {
    $indent = $before;
}

```

These special items will be processed at the top level, after all files have been read.

10b *<propagate header special items 10b>*≡ (11)

```

while (@chunkreflines and ref($chunkreflines[0])) {
    # propagate special items early, before processing lines
    push(@res, shift(@chunkreflines));
}

```

10c *<warn empty code chunk 10c>*≡ (11)

```

Carp::carp("WARNING: Code chunk <${chunkref}>> is" .
    " empty (at ${fname}:${lnum}).");

```

Defining `$line_new` will effectively replace the current line and search for more references. This allows more than one chunk reference per line.

```

11  <extract and process chunk reference 11>≡ (12b)
    my @chunkreflines = extract($prefix . $before, $fname,
        $lnum, $chunkref);
    <propagate header special items 10b>
    if (@chunkreflines > 1) {
        # many lines
        <setup $indent and $before_has_content 10a>
        my $first = shift(@chunkreflines);
        my $last = pop(@chunkreflines);
        if ($before_has_content or $first ne "\n") {
            push(@res, $before . $first);
        } else {
            push(@res, "\n");
        }
        for (@chunkreflines) {
            if (ref($_)) {
                # deal with special items later
                push(@res, $_);
            } elsif ($_ ne "\n") {
                push(@res, $indent . $_);
            } else {
                # no need to indent an empty line
                push(@res, "\n");
            }
        }
        if ($after) {
            chomp($last);
            $line_new = $indent . $last . $after . "\n";
        } elsif ($last ne "\n") {
            push(@res, $indent . $last);
        } else {
            # no need to indent an empty line
            push(@res, "\n");
        }
    } elsif (@chunkreflines) {
        # just 1 line
        if ($after) {
            chomp(my $tmp = $chunkreflines[0]);
            $line_new = $before . $tmp . $after . "\n";
        } else {
            <setup $before_has_content 9c>
            if ($before_has_content or $chunkreflines[0] ne "\n") {
                push(@res, $before . $chunkreflines[0]);
            }
        }
    }

```

```

        } else {
            push(@res, "\n");
        }
    }
} else {
    <warn empty code chunk 10c>
}

```

12a *<warn chunk does not exit 12a>*≡ (12b)
 Carp::carp("WARNING: Code chunk <\${chunkref}>>" .
 " does not exist (at \${fname}:\${lnum}).");

12b *<process line 12b>*≡ (13b)

```

my $found_chunk_ref;
for (;;) {
    $found_chunk_ref = ($line =~ /^(|.*?[^\@])<<(.?*[^^\@])>>(.*)/);
    if ($found_chunk_ref) {
        my $before = $1;
        my $chunkref = $2;
        my $after = $3;
        if ($chunks_options{$chunkref}) {
            my $line_new = undef;
            <extract and process chunk reference 11>
            if (defined($line_new)) {
                $line = $line_new;
                next;
            }
        } else {
            <warn chunk does not exit 12a>
        }
        $lnum = -1; # force line number directive
    } else {
        $line =~ s/\@(<<|>>)/$1/g;
        push(@res, $line);
    }
}
last;

```

A chunk referenced at least once is definitely a non-root chunk, so we need to render all chunks at most once

12c *<remove chunk data if listing roots 12c>*≡ (13b)

```

if ($list_roots and @input) {
    $chunks{$chunk} = [$input[0], $input[1], ""];
}

```

Special item 0 is a line number directive.

12d *<push line number directive 12d>*≡ (13b)

```

push(@res, [0, $fname, $lnum]);

```

```

13a  <error recursive chunk definition 13a>≡ (13b)
      Carp::croak("ERROR: Code chunk <<$chunk>> is used in its" .
        " own definition (at ${parent_fname}:${parent_lnum}).");

13b  <sub extract 13b>≡ (6)
      my %being_extracted = ();
      sub extract {
        my ($prefix, $parent_fname, $parent_lnum, $chunk) = @_;
        if ($being_extracted{$chunk}) {
          <error recursive chunk definition 13a>
        }
        $being_extracted{$chunk}++;
        $chunks_options{$chunk}->{nextextractions}++;
        my @res = ();
        my @input = get_many($chunks{$chunk});
        <remove chunk data if listing roots 12c>
        my $lnum_previous = -1;
        while (my ($fname, $lnum, $line) = splice(@input, 0, 3)) {
          if ($sync and $lnum != $lnum_previous + 1) {
            <push line number directive 12d>
          }
          <process line 12b>
          $lnum_previous = $lnum;
        }
        $being_extracted{$chunk}--;
        return @res;
      }

13c  <sub usage 13c>≡ (6)
      sub usage {
        my $arg = shift @_;

        print STDERR <<EOF;
        Unknown command line argument "$arg". See the manual page for help
        which is also included at the end of this Perl script.

        nofake [-Rchunk ...] [-L[format]] [file] ...
        nofake [--version | -v]
        nofake [--list-all] [file] ...
        nofake [--list-roots] [file] ...
      EOF
    }

13d  <error won't load from tty 13d>≡ (14d)
      Carp::croak("ERROR: Will not load binary data from a terminal.");

```

- 14a *<error won't dump to tty 14a>*≡ (14b)
 Carp::croak("ERROR: Will not dump binary data to a terminal.");
 See <https://perldoc.perl.org/5.8.9/Storable> for more information.
- 14b *<dump state 14b>*≡ (16)

```
use Storable qw{lock_store store_fd};
my $state = [ \%chunks_options, \%chunks];
if ($dump eq '-') {
    if (-t STDOUT) {
        <error won't dump to tty 14a>
    } else {
        store_fd($state, \*STDOUT);
    }
} else {
    lock_store($state, $dump);
}
```
- 14c *<error state file does not exist 14c>*≡ (14d)
 Carp::croak("ERROR: State file does not exist.");
- 14d *<load state 14d>*≡ (16)

```
use Storable qw{lock_retrieve fd_retrieve};
my $state;
if ($load eq '-') {
    if (-t STDIN) {
        <error won't load from tty 13d>
    } else {
        $state = fd_retrieve(*STDIN);
    }
} elsif (-f $load) {
    $state = lock_retrieve($load);
} else {
    <error state file does not exist 14c>
}
%chunks_options = %{ $state->[0] };
%chunks = %{ $state->[1] };
```
- 14e *<read files 14e>*≡ (16)

```
if ($load ne '-' and not @files) {
    push(@files, '-');
}
read_file($_) for @files;
```
- 14f *<arguments ignored for compatibility with notangle 14f>*≡ (15d)

```
elsif (/^-filter$/) { shift(@ARGV) }
```

- 15a *<arguments handling: dump and load 15a>*≡ (15d)
 elif (/^--dump=(.+)\$/) { \$dump = \$1 }
 elif (/^--dump\$/) { \$dump = shift(@ARGV) }
 elif (/^--load=(.+)\$/) { \$load = \$1 }
 elif (/^--load\$/) { \$load = shift(@ARGV) }
- 15b *<arguments handling: listing and no-op 15b>*≡ (15d)
 elif (/^--list-all\$/) { \$list_all = 1 }
 elif (/^--list-roots\$/) { \$list_roots = 1 }
 elif (/^--no-op\$/) { \$no_op = 1 }
- 15c *<arguments handling: -L, -R and version 15c>*≡ (15d)
 if (/^L(.+)/) {
 \$sync = 1;
 \$lineformat = \$1 if \$1;
 }
 elif (/^R(.+)/) { push(@chunks, \$1) }
 elif (/^-(v|-version)\$/) { version(); exit(0) }
- 15d *<process arguments 15d>*≡ (16)
<arguments handling: -L, -R and version 15c>
<arguments handling: listing and no-op 15b>
<arguments handling: dump and load 15a>
<arguments ignored for compatibility with notangle 14f>
 elif (/^(-.*)\$/) { usage(\$1); exit(1) }
 else { push(@files, \$_) }
- 15e *<error stdout output ambiguity 15e>*≡ (16)
 Carp::croak("ERROR: Cannot write to stdout due to output ambiguity.");

We parse the command line and start working: reading the noweb files and then extracting each chunk in the `chunks` variable.

```

16  <process command line and extract chunk 16>≡ (6)
    my @chunks = ();
    my @files = ();
    my $list_all = 0;
    my $no_op = 0;
    my $dump = 0;
    my $load = 0;

    while ($_ = shift(@ARGV)) {
        <process arguments 15d>
    }

    if ($dump eq '-') {
        if ($list_all or $list_roots or @chunks) {
            <error stdout output ambiguity 15e>
        }
        $no_op = 1;
    }

    if ($load) {
        <load state 14d>
    }

    <read files 14e>

    if ($dump) {
        <dump state 14b>
    }

    if ($no_op) {
        # useful for documentation chunks validation and set when dumping to
        # stdout
    } elsif ($list_all) {
        print("<<${_}>>") for sort(keys(%chunks_options));
    } elsif ($list_roots) {
        <output roots 17d>
    } else {
        <output lines 17c>
    }

```



```

17a  <output lines of $chunk 17a>≡ (17c)
      local $\ = undef;
      my ($first_fname, $first_lnum) = get_many($chunks{$chunk});
      for my $item (extract('', $first_fname, $first_lnum, $chunk)) {
          if (ref($item) eq 'ARRAY') {
              print line_directive($item->[1], $item->[2]);
          } else {
              print $item;
          }
      }

17b  <verify $chunk is defined 17b>≡ (17c)
      if (!$chunks_options{$chunk}) {
          Carp::croak("ERROR: The root chunk <<${chunk}>> was" .
              " not defined.");
      }

17c  <output lines 17c>≡ (16)
      if (not @chunks) {
          push(@chunks, '*');
      }
      for my $chunk (@chunks) {
          <verify $chunk is defined 17b>
      }
      for my $chunk (@chunks) {
          <output lines of $chunk 17a>
      }

17d  <output roots 17d>≡ (16)
      my @all_chunks = sort(keys(%chunks_options));
      # first pass, extract all chunks
      for my $chunk (@all_chunks) {
          my $opts = $chunks_options{$chunk};
          my ($first_fname, $first_lnum) = get_many($chunks{$chunk});
          extract('', $first_fname, $first_lnum, $chunk);
      }
      # second pass, report
      for my $chunk (@all_chunks) {
          my $opts = $chunks_options{$chunk};
          if ($opts->{nextextractions} == 1) {
              print("<<${chunk}>>");
          }
      }

```

```

18  <many utils 18>≡ (6)
    sub set_many (\%@) {
        my ($h, $k, @rest) = @_;
        if (defined($h->{$k})) {
            if (ref($h->{$k}) eq 'ARRAY') {
                push(@{ $h->{$k} }, @rest);
            } else {
                $h->{$k} = [$h->{$k}, @rest];
            }
        } elsif (@rest > 1) {
            $h->{$k} = [ @rest ];
        } elsif (@rest) {
            $h->{$k} = $rest[0];
        } else {
            Carp::carp("no value specified");
        }
    }
    sub get_many {
        # () yield undef in scalar context

        # return () instead of (undef)
        return () unless defined($_[0]);

        # return @{arg}
        return @{ $_[0] } if ref($_[0]) eq 'ARRAY';

        # return (arg)
        @_;
    }

```

2 nofake.bat

This was taken from perl\5.8.3\bin\runperl.bat and tested under Windows XP SP3 (x86). Simply extract it to somewhere in the PATH, see documentation below in DESCRIPTION section.

```
19 <nofake.bat 19>≡
  @rem = '---Perl---
  @echo off
  if "%OS%" == "Windows_NT" goto WinNT
  perl.exe -x -S "%0" %1 %2 %3 %4 %5 %6 %7 %8 %9
  goto endofperl
:WinNT
  perl.exe -x -S %0 %*
  if NOT "%COMSPEC%" == "%SystemRoot%\system32\cmd.exe" goto endofperl
  if %errorlevel% == 9009 echo You do not have Perl in your PATH.
  if errorlevel 1 goto script_failed_so_exit_with_non_zero_val 2>nul
  goto endofperl
  @rem ';
  #!perl -w
  #line 15
  $0 =~ s|\.\bat||i;
  unless (-f $0) {
    $0 =~ s|.*[/\\\]|;
    for (".", split ';', $ENV{PATH}) {
      $_ = "." if $_ eq "";
      $0 = "$_/$0" , goto doit if -f "$_/$0";
    }
    die "'$0' not found.\n";
  }
  doit: exec "perl.exe", "-x", $0, @ARGV;
  die "Failed to exec '$0': $!";
__END__

=head1 NAME

runperl.bat - "universal" batch file to run perl scripts

=head1 SYNOPSIS

    C:\> copy runperl.bat foo.bat
    C:\> foo
    [..runs the perl script 'foo'..]

    C:\> foo.bat
    [..runs the perl script 'foo'..]
```

=head1 DESCRIPTION

This file can be copied to any file name ending in the ".bat" suffix. When executed on a DOS-like operating system, it will invoke the perl script of the same name, but without the ".bat" suffix. It will look for the script in the same directory as itself, and then in the current directory, and then search the directories in your PATH.

It relies on the C<exec()> operator, so you will need to make sure that works in your perl.

This method of invoking perl scripts has some advantages over batch-file wrappers like C<pl2bat.bat>: it avoids duplication of all the code; it ensures C<\$0> contains the same name as the executing file, without any egregious ".bat" suffix; it allows you to separate your perl scripts from the wrapper used to run them; since the wrapper is generic, you can use symbolic links to simply link to C<runperl.bat>, if you are serving your files on a filesystem that supports that.

On the other hand, if the batch file is invoked with the ".bat" suffix, it does an extra C<exec()>. This may be a performance issue. You can avoid this by running it without specifying the ".bat" suffix.

Perl is invoked with the -x flag, so the script must contain a C<#!perl> line. Any flags found on that line will be honored.

=head1 BUGS

Perl is invoked with the -S flag, so it will search the PATH to find the script. This may have undesirable effects.

=head1 SEE ALSO

perl, perlwin32, pl2bat.bat

=cut

__END__
:endofperl

3 nofake.sh

A shell script wrapper upon `nofake` that conditionally write or update an output file. The main goal is to integrate nicely with makefiles.

```
21 <nofake.sh 21>≡
    #!/bin/sh

    set -eu

    if [ x"${ZSH_VERSION:-}" != x ]; then
        # let zsh behave like dash, bash and others shells when
        # expanding parameters
        setopt sh_word_split
    fi

    # environment variables that can affect the result

    SRC_PREFIX=${SRC_PREFIX:-}
    NOFAKE=${NOFAKE:-nofake}
    NOFAKEFLAGS=${NOFAKEFLAGS:-}
    ECHO=${ECHO:-echo}
    MV=${MV:-mv -f}
    RM=${RM:-rm -f}
    TOUCH=${TOUCH:-touch}
    CHMOD=${CHMOD:-chmod 0444}
    CMP=${CMP:-cmp -s}

    opts=
    chunks=
    sources=
    output=

    while [ $# -gt 0 ]; do
        case "${1}" in
            -L*) opts="${opts} '${1}'" ;;
            -R*) chunks="${chunks} '${1}'" ;;

            -o|--output) output=${2}; shift ;;
            --output=*) output=${1#*=} ;;
            -o*) output=${1#??} ;;

            -) sources="${sources} '-'" ;;
            -*)
                echo "${0##*/}: Unrecognized option '${1}'." 1>&2
                exit 1
        esac
        shift
    done
```

```

        ;;
        *) sources="${sources} '${SRC_PREFIX}${1}'" ;;
    esac
    shift
done

if [ x"${output}" = x ]; then
    echo "${0##*/}: No output specified, use '-o' option." 1>&2
    exit 1
fi

stamp=${output}.stamp
tmpfile=.tmp.${output}

uptodate(){
    test="test '(' -e '${stamp}' -a -e '${output}'"
    eval "set -- ${sources}"
    for src do test="${test} -a '${stamp}' -nt ${src}"; done
    test="${test} ')"
    eval "${test}"
}

if ! uptodate; then
    eval "set -- ${opts} ${chunks} ${sources}"
    ${NOFAKE} ${NOFAKEFLAGS} "$@" >"${tmpfile}"
    if ! ${CMP} "${tmpfile}" "${output}"; then
        ${ECHO} "Generate ""'${output}'""."
        ${MV} "${tmpfile}" "${output}"
        ${CHMOD} "${output}"
    else
        ${RM} "${tmpfile}"
        ${ECHO} 'Output ""'${output}"" did not change.'
    fi
    ${TOUCH} "${stamp}"
else
    ${ECHO} 'Output ""'${output}"" is up to date.'
fi

```

4 List of all chunks

<arguments handling: -L, -R and version 15c>
 <arguments handling: dump and load 15a>
 <arguments handling: listing and no-op 15b>
 <arguments ignored for compatibility with notangle 14f>
 <defaults 7>
 <dump state 14b>
 <error recursive chunk definition 13a>
 <error state file does not exist 14c>
 <error stdout output ambiguity 15e>
 <error won't dump to tty 14a>
 <error won't load from tty 13d>
 <extract and process chunk reference 11>
 <line directive 8a>
 <load state 14d>
 <manual page in POD format 2>
 <many utils 18>
 <nofake 6>
 <nofake.bat 19>
 <nofake.sh 21>
 <output lines 17c>
 <output lines of \$chunk 17a>
 <output roots 17d>
 <process arguments 15d>
 <process command line and extract chunk 16>
 <process line 12b>
 <propagate header special items 10b>
 <push line number directive 12d>
 <read file 9b>
 <read files 14e>
 <read line 9a>
 <read line vars 8c>
 <remove chunk data if listing roots 12c>
 <setup \$before_has_content 9c>
 <setup \$indent and \$before_has_content 10a>
 <sub extract 13b>
 <sub usage 13c>
 <verify \$chunk is defined 17b>
 <warn chunk does not exit 12a>
 <warn empty code chunk 10c>
 <warn unescaped << in documentation chunk 8b>