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

February 11, 2024

Contents

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] [B<--error>] [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

=item B<--error>

Treat warnings as errors.

=back

=head1 SYNTAX OF NOWEB FILES

The authoritive 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 recursivley 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.harard.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";

my $carp_or_croak = \&Carp::carp;

sub version {
    print <<'EOF';
    (c) 2023 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⟩

⟨*utils* 19⟩
⟨*line directive* 8a⟩
⟨*read file* 9a⟩
⟨*sub extract* 13d⟩
⟨*sub usage* 14b⟩

⟨*process command line and extract chunk* 17⟩

```
__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 ⟨*line directive* 8a⟩≡ (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 at sign (`@`) followed by a space or newline.

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

8b ⟨*read line vars* 8b⟩≡ (9a)

```
my $chunk = '';
```

8c ⟨*read line* 8c⟩≡ (9a)

```
if ($line =~ m{^ << (.+?) >>= \s* $}x) {
    $chunk = $1;
    if (!$chunks_options{$chunk}) {
        $chunks_options{$chunk} = {nextractions => 0};
    }
    $chunks_options{$chunk}->{nchunks}++;
} elsif ($chunk) {
    if ($line =~ m{^ \@ (?: $ | \s) }x) {
        $chunk = '';
    } else {
        # regular line inside chunk
        $line =~ s,^\@\@,\@,;
        set_many(%chunks, $chunk, $ARGV, int($.), $line . "\n");
    }
}
```

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

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.

9b ⟨*setup $before_has_content* 9b⟩≡ (9c 10)
```
my $before_has_content = $before !~ m{^\s*$};
```

9c ⟨*setup $indent and $before_has_content* 9c⟩≡ (10)
```
my $indent;
⟨setup $before_has_content 9b⟩
if ($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.

9d ⟨*propagate header special items* 9d⟩≡ (10)
```
while (@chunkreflines and ref($chunkreflines[0]) eq 'ARRAY') {
    # propagate special items early, before processing lines
    if (@res and ref($res[-1]) eq 'ARRAY') {
        push(@{ $res[-1] }, @{ shift(@chunkreflines) });
    } else {
        push(@res, [ @{ shift(@chunkreflines) } ]);
    }
}
```

9e ⟨*warn empty code chunk* 9e⟩≡ (10)
```
$carp_or_croak->("WARNING: Code chunk <<${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.

10 ⟨*extract and process chunk reference* 10⟩≡ (12)

```
my @chunkreflines = extract($prefix . $before, $fname,
    $lnum, $chunkref);
⟨propagate header special items 9d⟩
if (@chunkreflines > 1) {
    # many lines
    ⟨setup $indent and $before_has_content 9c⟩
    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 $before_has_content 9b⟩
        if ($before_has_content or $chunkreflines[0] ne "\n") {
            push(@res, $before . $chunkreflines[0]);
```

```
            } else {
                push(@res, "\n");
            }
        }
    } else {
        ⟨warn empty code chunk 9e⟩
    }
```

11 ⟨warn chunk does not exit 11⟩≡ (12)

```
    $carp_or_croak->("WARNING: Code chunk <<${chunkref}>>" .
        " does not exist (at ${fname}:${lnum}).");
```

The chunk name usage follows the same rules from the definition, e.g., `<<<hi>>>=` defines a chunk named `<hi>`, so, the usage of `<<<hi>>>` also recall a chunk named `<hi>`, there is consistency. But there are cases like `<<<prefix>>page.html>` that requires special attention:

```
<<<prefix>>=
<http://not.great.not.terrible/
<<#include <stdio.h>>>=
#include <stdio.h> /* for comparison */
<<*>>=
<<<prefix>>page.html>
<<#include <stdio.h>>>
```

, the output will be:

```
<http://not.great.not.terrible/page.html>
#include <stdio.h> /* for comparison */
```

12 ⟨*process line* 12⟩≡ (13d)

```
my $found_chunk_ref;
for (;;) {
    $found_chunk_ref = ($line =~ m{^ (|.*?[^\@]) << (.*?[^\@]) >> (|[^>].*?) $}x);
    if ($found_chunk_ref) {
        my $before = $1;
        my $chunkref = $2;
        my $after = $3;
        if ($chunks_options{$chunkref}) {
            my $line_new = undef;
            ⟨extract and process chunk reference 10⟩
            if (defined($line_new)) {
                $line = $line_new;
                next;
            }
        } else {
            ⟨warn chunk does not exit 11⟩
        }
        $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

13a ⟨*remove chunk data if listing roots* 13a⟩≡ (13d)
```
if ($list_roots and @input) {
    $chunks{$chunk} = [$input[0], $input[1], ""];
}
```

Special item `0` is a line number directive.

13b ⟨*push line number directive* 13b⟩≡ (13d)
```
my $linenum = [0, $fname, $lnum];
if (@res and ref($res[$#res]) eq 'ARRAY') {
    push(@{ $res[$#res] }, $linenum);
} else {
    push(@res, [$linenum]);
}
```

13c ⟨*error recursive chunk definition* 13c⟩≡ (13d)
```
Carp::croak("ERROR: Code chunk <<$chunk>> is used in its" .
    " own definition (at ${parent_fname}:${parent_lnum}).");
```

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

14a ⟨*usage text* 14a⟩≡ (14b)

```
Usage:

    nofake [-Rchunk ...] [-L[format]] [--dump state] \
           [--load state] [--error] [file] ...

    nofake [--version | -v]

    nofake [--list-all] [file] ...

    nofake [--list-roots] [file] ...
```

14b ⟨*sub usage* 14b⟩≡ (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.

⟨usage text 14a⟩
EOF
}
```

14c ⟨*error won't load from tty* 14c⟩≡ (15a)

```
Carp::croak("ERROR: Will not load binary data from a terminal.");
```

14d ⟨*error won't dump to tty* 14d⟩≡ (14e)

```
Carp::croak("ERROR: Will not dump binary data to a terminal.");
```

See https://perldoc.perl.org/5.8.9/Storable for more information.

14e ⟨*dump state* 14e⟩≡ (17)

```
use Storable qw{lock_store store_fd};
my $state = [\%chunks_options, \%chunks];
if ($dump eq '-') {
    if (-t STDOUT) {
        ⟨error won't dump to tty 14d⟩
    } else {
        store_fd($state, \*STDOUT);
    }
} else {
    lock_store($state, $dump);
}
```

14f ⟨*error state file does not exist* 14f⟩≡ (15a)

```
Carp::croak("ERROR: State file does not exist.");
```

15a ⟨*load state* 15a⟩≡ (17)

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

15b ⟨*read files* 15b⟩≡ (17)

```
if ($load ne '-' and not @files) {
    push(@files, '-');
}
read_file($_) for @files;
```

15c ⟨*arguments ignored for compatibility with notangle* 15c⟩≡ (16a)

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

15d ⟨*arguments handling: dump and load* 15d⟩≡ (16a)

```
elsif (/^--dump=(.+)$/) { $dump = $1 }
elsif (/^--dump$/)      { $dump = shift(@ARGV) }
elsif (/^--load=(.+)$/) { $load = $1 }
elsif (/^--load$/)      { $load = shift(@ARGV) }
```

15e ⟨*arguments handling: force error on warnings* 15e⟩≡ (16a)

```
elsif (/^--error$/) { $carp_or_croak = \&Carp::croak; }
```

15f ⟨*arguments handling: listing and no-op* 15f⟩≡ (16a)

```
elsif (/^--list-all$/)   { $list_all = 1 }
elsif (/^--list-roots$/) { $list_roots = 1 }
elsif (/^--no-op$/)      { $no_op = 1 }
```

15g ⟨*arguments handling: -L, -R and version* 15g⟩≡ (16a)

```
if (/^-L(.*)/) {
    $sync = 1;
    $lineformat = $1 if $1;
}
elsif (/^-R(.+)/)         { push(@chunks, $1) }
elsif (/^-(v|-version)$/) { version(); exit(0) }
```

16a ⟨*process arguments* 16a⟩≡ (17)
```
⟨arguments handling: -L, -R and version 15g⟩
⟨arguments handling: listing and no-op 15f⟩
⟨arguments handling: dump and load 15d⟩
⟨arguments handling: force error on warnings 15e⟩
⟨arguments ignored for compatibility with notangle 15c⟩
elsif (/^(-.+)$/) { usage($1); exit(1) }
else { push(@files, $_) }
```

16b ⟨*error stdout output ambiguity* 16b⟩≡ (17)
```
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.

17 ⟨*process command line and extract chunk* 17⟩≡ (6)

```
my @chunks = ();
my @files = ();
my $list_all = 0;
my $no_op = 0;
my $dump = 0;
my $load = 0;

while ($_ = shift(@ARGV)) {
    ⟨process arguments 16a⟩
}

if ($dump eq '-') {
    if ($list_all or $list_roots or @chunks) {
        ⟨error stdout output ambiguity 16b⟩
    }
    $no_op = 1;
}

if ($load) {
    ⟨load state 15a⟩
}

⟨read files 15b⟩

if ($dump) {
    ⟨dump state 14e⟩
}

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 18d⟩
} else {
    ⟨output lines 18c⟩
}
```

18a ⟨*output lines of $chunk* 18a⟩≡ (18c)

```
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') {
        my $head = $item->[-1];
        print line_directive($head->[1], $head->[2]);
    } else {
        print $item;
    }
}
```

18b ⟨*verify $chunk is defined* 18b⟩≡ (18c)

```
if (!$chunks_options{$chunk}) {
    Carp::croak("ERROR: The root chunk <<${chunk}>> was" .
        " not defined.");
}
```

18c ⟨*output lines* 18c⟩≡ (17)

```
if (not @chunks) {
    push(@chunks, '*');
}
for my $chunk (@chunks) {
    ⟨verify $chunk is defined 18b⟩
}
for my $chunk (@chunks) {
    ⟨output lines of $chunk 18a⟩
}
```

18d ⟨*output roots* 18d⟩≡ (17)

```
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 ($opts->{nextractions} == 1) {
        print("<<${chunk}>>");
    }
}
```

19 ⟨*utils* 19⟩≡ (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.

20 ⟨*nofake.bat* 20⟩≡

```
@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 conditionaly write or update an output file. The main goal is to integrate nicely with makefiles.

22 ⟨*nofake.sh* 22⟩≡

```
#!/bin/sh

set -eu

die(){ ev=$1; shift; for msg in "$@"; do echo "${msg}"; done; exit "${ev}"; }

⟨zsh fix 25c⟩

# environment variables that can affect the result

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

⟨u0Aa 24a⟩
⟨r0Aa 24b⟩
⟨temporary_file 25a⟩
⟨rm_tmpfiles 25b⟩

opts=
chunks=
sources=
output=

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

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

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

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

stamp=${output}.stamp

args_id(){
    perl -MDigest::SHA=sha256_hex \
        -le'print(sha256_hex(join(q{&}, @ARGV)))' \
        -- "$@"
}

uptodate(){
    if [ ! -e "${stamp}" ]; then
        false
    else
        current_id=`head -n1 "${stamp}"`
        if [ ! -e "${output}" -o x"${sources_id}" != x"${current_id}" ]; then
            false
        else
            eval "set -- ${sources}"
            test=
            for src do test="${test:+${test} -a }'${stamp}' -nt ${src}"; done
            eval "test '(' ${test} ')'"
        fi
    fi
}

tmpfile=`temporary_file`
tmpfiles="${tmpfiles} '${tmpfile}'"

eval "set -- ${opts} ${chunks} ${sources}"
sources_id=`args_id "$@"`
```

```
if ! uptodate; then
    eval "set -- ${opts} ${chunks} ${sources}"
    ${NOFAKE} ${NOFAKEFLAGS} "$@" >"${tmpfile}"
    if ! ${CMP} "${tmpfile}" "${output}"; then
        ${ECHO_INFO} "Generate "'"'"${output}"'"'"." 1>&2
        ${MV} "${tmpfile}" "${output}"
        ${CHMOD} "${output}"
    else
        ${RM} "${tmpfile}"
        ${ECHO_INFO} 'Output "'"${output}"'" did not change.' 1>&2
    fi
    ${ECHO} "${sources_id}" > "${stamp}"
else
    ${ECHO_INFO} 'Output "'"${output}"'" is up to date.' 1>&2
fi
```

A random string using `/dev/urandom` is very useful.

24a ⟨*u0Aa* 24a⟩≡ (22)

```
u0Aa(){
    perl -e'@map=map{chr}48..57,65..90,97..122;
        $c = $ARGV[0];
        while($c and read(STDIN,$d,64)){
            for $x (unpack(q{C*},$d)) {
                last unless $c;
                next if $x >= scalar(@map);
                $c--;
                print($map[$x]);
            }
        }' -- "${1}" </dev/urandom
}
```

A fallback from `u0Aa` when `/dev/urandom` is not available, it seems to use time and process id as seed, at least, after perl 5.004 (really old).

24b ⟨*r0Aa* 24b⟩≡ (22)

```
r0Aa(){
    perl -e'@map=map{chr}48..57,65..90,97..122;
        sub c{$map[int(rand(scalar(@map)))]}
        for ($c=$ARGV[0];$c;$c--) { print(c) }' -- "${1}"
}
```

25a ⟨*temporary_file* 25a⟩≡ (22)

```
temporary_file(){
    if type mktemp >/dev/null 2>&1; then
        tmpfile=‘mktemp‘
    elif type perl >/dev/null 2>&1 && test -r /dev/urandom; then
        tmpfile="/tmp/tmp.‘u0Aa 12‘"
        ( umask 0177; : > "${tmpfile}" )
    elif type perl >/dev/null 2>&1; then
        tmpfile="/tmp/tmp.‘r0Aa 12‘"
        ( umask 0177; : > "${tmpfile}" )
    else
        die 1 ’error: mktemp not found’
    fi
    echo "${tmpfile}"
}
```

25b ⟨*rm_tmpfiles* 25b⟩≡ (22)

```
tmpfiles=
rm_tmpfiles(){
    eval "set -- ${tmpfiles}"
    rm -f -- "$@"
}
# 0:exit, 1:hup, 2:int, 3:quit, 15:term
trap ’rm_tmpfiles’ 0 1 2 3 15
```

25c ⟨*zsh fix* 25c⟩≡ (22)

```
if [ x"${ZSH_VERSION:-}" != x ]; then
    # let zsh behave like ash, ksh and other standard
    # shells when expanding parameters
    setopt sh_word_split
fi
```

4 List of all chunks

⟨*warn chunk does not exit* 11⟩
⟨*warn empty code chunk* 9e⟩
⟨*zsh fix* 25c⟩