Literate Programming using `noweb`[*]

Andrew L. Johnson and Brad C. Johnson

December 19, 2000

Introduction

> Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *humans* what we want the computer to do. (Donald E. Knuth, 1984).

In essence, this is the purpose of literate programming (LP for short). Such an environment reverses the notion of including documentation, in the form of comments, within the code, to one where the code is embedded within a program's description. In doing so, literate programming facilitates the development and presentation of computer programs that more closely follow the conceptual map from the problem space to the solution space. This, in turn, leads to programs that are easier to debug and maintain.

When literate programming, one specifies the program description and the program code in a single source file in the order best suited to human understanding. The program code can be extracted and assembled into a form understandable for the compiler or interpreter by a process called 'tangling'. Documentation is produced by a process of 'weaving' the description and code into a form ready to be typeset (most often by TEXor LATEX).

Many different tools have been created for literate programming over the years and most of the more popular are based, either directly or conceptually, on the `WEB` system created by D. E. Knuth [*cf.* 1984. Literate Programming. *The Computer Journal* (27)2:97-111]. This article focuses on Norman Ramsey's `noweb`—a simple to use, extensible literate programming tool that is independent of the target programming language.

Overview of the noweb System.

When you write a literate program using `noweb` you create a simple text file (which by convention has a .nw extension) in which you provide all of the tech-

[*]A version of this article has been previously published in the Linux Journal (issue 42, 1997), and is made available here with permission of the Linux Journal

nical documentation for the various parts of the program along with the actual source code for each part of the program.

This file (see Listing 1), which we will refer to as the `nw` source file, is then processed by `noweave` to create the documentation in a form ready for typesetting (the 'typeset version' of the program; see Fig 1),or processed by `notangle` to extract the code chunks and assemble them in their proper order for the compiler or interpreter (the 'executable version' of the program; see Listing 2). These two processes are not stand-alone programs, but a set of filters through which the `nw` source file is piped. It is this pipeline system that makes `noweb` both flexible and extensible as the pipelines can be modified and new filters can be created and inserted in the pipelines to change the behavior of `noweb`.

Listing 1: ***The Literate Source***

```
\documentclass[10pt]{article}
\usepackage{noweb}
\noweboptions{smallcode,longchunks}

\begin{document}
\pagestyle{noweb}

@ \paragraph{Introduction}
This is [[autodefs.perl]]\footnote{Copyright 1997, Andrew L.
Johnson and Brad C. Johnson,  All rights reserved.},
a Perl script to be used as an [[autodefs]] filter
in the [[noweb]] pipeline to identify and index
some common Perl definitions.  Since this
file is also meant to show off some of the
features of [[noweb]] it is purposely verbose
and contorted.

Perl does not require the formal declaration or typing of
variables which makes it difficult to
differentiate between declarations and usages of
variables.  We may however find definitions of [[sub]]'s and
[[package]]'s with little difficulty and that is the purpose of
this module.  Before we begin we need to know
some facts about [[noweb]]'s pipeline structure.\footnote{Noweb's
pipeline structure is described in the \textit{Noweb Hackers
Guide} which is included in the [[noweb]] distribution.}
Actual code in the pipeline lie between lines
of the form [[@begin code]] and [[@end code]].
In Perl these are easily recognized by the following regular
expressions.
<<Global variables>>=
$begin_code_pat = "^\@begin code";
$end_code_pat   = "^\@end code";
@ %def $begin_code_pat $end_code_pat
@ Within a code block there are many types of lines.  Ones
that contain actual code are prefixed by [[@text]].
<<Global variables>>=
$code_line_pat = "^\@text";
@ %def $code_line_pat

@ If, on a code line inside a code block, we find something that
should be added to the ``Defines'' block at the end of the code
chunk and appear in the index, then we need to add a line to the
pipeline of the form ``[[@index defn <ident>]]''.
<<Global variables>>=
$index_prefix = "\@index defn";
@ %def $index_prefix

@ \paragraph{autodefs.perl}
Our actual Perl script has the following simple shape:
<<autodefs.perl>>=
#!/usr/bin/perl
<<Global variables>>
<<[[process_code_chunk]] subroutine>>
while ( <> ) {
    print $_;
    if (/$begin_code_pat/o) {
        process_code_chunk;
    }
}
@

\paragraph{Processing the code chunk}
To process the code chunk we need to perform a few housekeeping
tasks.  First, we only want to consider lines that begin with
[[$code_line_pat]] and second, we want to stop when we find a line
that matches [[$end_code_pat]].  The following loop will suffice
for this purpose.
<<[[process_code_chunk]] subroutine>>=
sub process_code_chunk {
    while ( ($_ = <>) && !/$end_code_pat/o ) {
        print $_;
        if( /$code_line_pat/o ) {
           <<Find and print any definitions>>
        }
    }
    print $_; # make sure we print the ``@end code'' line
}
@
@ When checking for definitions we first strip off
any comments since [[sub]] or [[package]] may
also occur in a comment.  We then build
a list [[@def_list]] which contain all of the
```

```
[[sub]] and [[package]] definitions on the line
and print out an [[@index defn]] line for
each.
<<Find and print any definitions>>=
$_ =~ s/#.*//o;
@def_list = (/sub\s(\w+)/go, /package\s(\w+)/go);
foreach $item (@def_list) {
    print "$index_prefix $item\n";
}
@

\paragraph{Defined Chunks}\par\noindent

\nowebchunks

\paragraph{Index}\par\noindent

\nowebindex
@
\end{document}
```

Listing 2: ***The Tangled Code***

```
#!/usr/bin/perl
$begin_code_pat = "^\@begin code";
$end_code_pat   = "^\@end code";
$code_line_pat = "^\@text";
$index_prefix = "\@index defn";
sub process_code_chunk {
    while ( ($_ = <>) && !/$end_code_pat/o ) {
        print $_;
        if( /$code_line_pat/o ) {
           $_ =~ s/#.*//o;
           @def_list = (/sub\s(\w+)/go, /package\s(\w+)/go);
           foreach $item (@def_list) {
               print "$index_prefix $item\n";
           }
        }
    }
    print $_; # make sure we print the ‘‘@end code’’ line
}
while ( <> ) {
    print $_;
    if (/$begin_code_pat/o) {
        process_code_chunk;
    }
}
```

Like most literate programming tools, `noweb` depends on TEXor LATEX—(La)TEXto refer to either—for typesetting the documentation (although it has options for producing html output as well). However, one need not be a (La)TEXguru to produce good results as all of the hard work of cross-referencing, indexing, and typesetting the code is handled automatically by `noweave`.

The Typeset Documentation.

The best way to get a feel for the capabilities of `noweb` is by reference to the finished product—that is, the typeset version of a program. Figure 1 represents the typeset version of a perl script which actually extends `noweb`'s functionality by providing a limited 'autodefs' filter for recognizing and marking package and subroutine names in Perl for automatic cross-referencing and indexing.

Figure 1

Introduction This is `autodefs.perl`[1], a Perl script to be used as an `autodefs` filter in the `noweb` pipeline to identify and index some common Perl definitions. Since this file is also meant to show off some of the features of `noweb` it is purposely verbose and contorted.

Perl does not require the formal declaration or typing of variables which makes it difficult to differentiate between declarations and usages of variables. We may however find definitions of `sub`'s and `package`'s with little difficulty and that is the purpose of this module. Before we begin we need to know some facts about `noweb`'s pipeline structure.[2] Actual code in the pipeline lie between lines of the form `@begin code` and `@end code`. In Perl these are easily recognized by the following regular expressions.

5a ⟨*Global variables* 5a⟩≡ (6b) 5b▷

```
$begin_code_pat = "^\@begin code";
$end_code_pat   = "^\@end code";
```

Defines:
`$begin_code_pat`, *used in chunk 6b.*
`$end_code_pat`, *used in chunk 6c.*

Within a code block there are many types of lines. Ones that contain actual code are prefixed by `@text`.

5b ⟨*Global variables* 5a⟩+≡ (6b) ◁5a 6a▷

```
$code_line_pat = "^\@text";
```

Defines:
`$code_line_pat`, *used in chunk 6c.*

[1]Copyright 1997, Andrew L. Johnson and Brad C. Johnson, All rights reserved.
[2]Noweb's pipeline structure is described in the *Noweb Hackers Guide* which is included in the `noweb` distribution.

If, on a code line inside a code block, we find something that should be added to the "Defines" block at the end of the code chunk and appear in the index, then we need to add a line to the pipeline of the form "`@index defn <ident>`".

6a ⟨*Global variables* 5a⟩+≡ (6b) ◁5b
```
  $index_prefix = "\@index defn";
```
Defines:
`$index_prefix`, *used in chunk 7.*

autodefs.perl Our actual Perl script has the following simple shape:

6b ⟨*autodefs.perl* 6b⟩≡
```
  #!/usr/bin/perl
  ⟨Global variables 5a⟩
  ⟨process_code_chunk subroutine 6c⟩
  while ( <> ) {
      print $_;
      if (/$begin_code_pat/o) {
          process_code_chunk;
      }
  }
```
Uses `$begin_code_pat` *5a and* `process_code_chunk` *6c.*

Processing the code chunk To process the code chunk we need to perform a few housekeeping tasks. First, we only want to consider lines that begin with `$code_line_pat` and second, we want to stop when we find a line that matches `$end_code_pat`. The following loop will suffice for this purpose.

6c ⟨`process_code_chunk` *subroutine* 6c⟩≡ (6b)
```
  sub process_code_chunk {
      while ( ($_ = <>) && !/$end_code_pat/o ) {
          print $_;
          if( /$code_line_pat/o ) {
              ⟨Find and print any definitions 7⟩
          }
      }
      print $_; # make sure we print the ‘‘@end code’’ line
  }
```
Defines:
`process_code_chunk`, *used in chunk 6b.*
Uses `$code_line_pat` *5b and* `$end_code_pat` *5a.*

When checking for definitions we first strip off any comments since `sub` or `package` may also occur in a comment. We then build a list `@def_list` which contain all of the `sub` and `package` definitions on the line and print out an `@index defn` line for each.

7 ⟨*Find and print any definitions* 7⟩≡ (6c)

```
  $_ =~ s/#.*//o;
  @def_list = (/sub\s(\w+)/go, /package\s(\w+)/go);
  foreach $item (@def_list) {
      print "$index_prefix $item\n";
  }
```

Uses `$index_prefix` *6a.*

Defined Chunks

Index

When looking at this example one can quickly see how chunks of actual code are interspersed throughout the descriptive text. Each code chunk is uniquely identified by page number and an alphabetic sub-page reference. For example, in Figure 1, there are four code chunks on the first page labeled in the left margin as 1a, 1b, 1c, and 1d.

Besides the marginal tag, the first line of each code chunk also has its name and a chunk reference enclosed in angle brackets at the left margin and perhaps cross-reference information at the right margin. Lets examine chunk 1b more closely—a reasonable facsimile of its first line is:

1b $\langle Global variables 1a\rangle{+}\equiv$ 1d $\triangleleft$ 1a 1b $\triangleright$

This line tells us that we are now in chunk 1b. The '$\langle$ *Global variables 1a*$\rangle{+}\equiv$' construct tells us we are working on the chunk named 'Global variables' whose definition begins in chunk 1a. The '$+\equiv$' indicates that we are adding to the definition of 'Global variables'. At the right margin we encounter '(1d) ¡1a 1c¿', which means that the chunk we are defining is used in chunk 1d, and that the current chunk is continued from chunk 1a and will be further continued in chunk 1c. It should be noted that all of these visual cross-referencing clues—with the exception of the chunk name itself—are provided automatically by `noweb`.

At the end of any chunk there are two optional footnotes— a 'Defines' footnote and a 'Uses' footnote. A user can manually specify, in the `nw` source file, a list of identifiers (i.e., variables or subroutines) which are defined in the current chunk. Some such identifiers may be automatically recognized if an 'autodefs' filter for the programming language is used (there are autodefs filters available for many languages including C, Icon, TEX, yacc and pascal). These identifiers are listed in the 'Defines' footnote below the chunk where their definition occurs, along with a reference to any chunks which use them. Any occurrence of a defined identifier is referenced in a 'Uses' footnote below the chunk the chunk that uses that identifier.

For example, in Figure 1, we see that chunk 1c defines the term `$index_prefix` which is used in chunk 2b. A quick peek at chunk 2b verifies that, indeed, this term is used and appears in the 'Uses' footnote for that chunk.

Chunk 1d, named 'autodefs.perl', represents the top level description of our entire program. This chunk is referred to as a 'root' chunk in `noweb` and is not used in any other chunk. Our example has but one root chunk, though you may define as many as you wish in your `nw` source file and `notangle` can extract each of them into separate files.

The first line of code in chunk 1d is the obligatory `#!/usr/bin/perl` line which must begin all perl scripts intended to be invoked as an executable program. However, the next two lines are not lines of perl code at all but instead are references to other named chunk definitions. Such references indicate that the code from those referenced chunks will be inserted at this point in the executable program extracted by `notangle`. Thus we have a broad overview of our program uncluttered by the specific global variable initializations and subroutine definitions.

Looking at chunk 2a, which is included in our root chunk, we see that it also includes another chunk, chunk 2b. This demonstrates that the inclusion of chunks can be nested (to practically any level) and can occur in any order in the documentation (definitions need not preceed uses).

Our documentation ends with two optional indices provided by `noweb`—an index of code chunks and an index of idenfiers.

Writing the Program in `noweb`.

With the knowledge of what comes out the end of the pipeline in hand, we can now describe the structure of the `nw` source file itself. The `nw` source file for our example program is given in Listing 1.

When you write your `noweb` program you alternate between explaining some piece of code and providing the formal definition of that piece of code. You must indicate whether you are entering documentation or code by use of two `noweb` tags.

To begin writing documentation one starts with an @ symbol in the left column followed by either a space or a newline. This indicates that all of text which follows, at least up to the next tag, is documentation text. All of the text which occurs in documentation text is passed through the filtering process to the (La)TEXfile. Thus, the author is responsible for providing any special formatting such as sections, tables, footnotes, and mathematical formulae which may be desired or needed in the documentation. In addition to the standard (La)TeX command set, `noweb` provides three additional control sequences: Any text surrounded by double square brackets in the text is typeset in the same fashion as literal code; and the
nowebindex and
nowebchunks commands expand into the two types of indices shown at the end of our example in Figure 1.

To indicate the beginning of a code chunk you simply use double angle brackets surrounding a name for the code chunk and followed by an equal sign:

```
<<code_chunk_name>>=
```

Everything following this construct is considered to be literal code, or a reference to another chunk name. You reference another chunk name by placing its name in double angle brackets with no trailing equal sign. As with documentation, a code chunk is terminated when another tag is encountered. To continue a code chunk definition you simply start a new code chunk using the same name within the brackets as the chunk you are continuing from.

The special formatting and cross-referencing of code chunks is handled automatically by `noweb` and requires no special input by the user—with the exception of manually specifying identifier definitions.

To manually indicate a list of identifiers which are defined in a given chunk you terminate that chunk with a line of the form:

```
@ %def ident1 ident2
```

The identifiers given on the line will be placed in a 'Defines' footnote for that chunk and will automatically be cross-referenced and indexed by `noweb` as described in the previous section.

The process by which `notangle` extracts the code into a form suitable for the compiler or interpreter follows just a few simple rules. A root chunk is specified on the command line as the chunk to be extracted and assembled. This chunk is then output line by line until a reference to another chunk is encountered. At this point, the referenced chunk is output line by line—and similarly for any chunks referenced therein. When the referenced chunk has been output the process of outputting the root chunk continues.

When dealing with continued chunks—two or more chunks sharing the same name...`notangle` concatenates their definintions in order of appearance into a single named chunk. The extracted code for our example program is in Listing 2, and it can be seen that all spacing and indentation is preserved appropriately in the executable version.

It is this extraction and assembly process of `notangle` that allows the explanation of the program and the presentation of each part of the program to proceed in an order independent of how the program must be ordered for the compiler or interpreter.

The Incantations.

Now that we know to create a program in `noweb` we can examine the methods of generating our typeset and executable versions of the program. The `noweb` distribution provides a general shell script called, remarkably, `noweb` which drives the `notangle` and `noweave` processes. However, this method of invocation, though simple, is somewhat limited. We will focus here on using each tool separately as this provides a more flexible approach.

When you invoke `notangle` you specify a chunk name (a root chunk) to extract and assemble from the `nw` source file. If you fail to specify a chunk, `notangle` will search for a chunk named '*' to extract (this is the default root chunk in a `noweb` program). The `notangle` tool writes to *stdout* so you must redirect this to a file of your choice. The general form of the command is:

```
notangle [-Rroot_chunk] [-Lformat]
         [-filter cmd] source.nw > programfile
```

Thus, to extract the executable version of our example program we used:

```
notangle -Rautodefs.perl autodefs.perl.nw > autodefs.perl
```

The `-R` option specifies which root chunk to extract. The `-L` option is used to embed line directives, if they are supported by the compiler/interpreter you will be using. The line directives refer to locations in the `nw` source file, thus, when debugging your code you need not ever refer to the executable version, rather you can simply edit the code in the `nw` source file. The default format of the line directives is for use with the C preprocessor, but also work well with Perl with one catch. The line directives are emitted whenever a chunk is entered or returned to, and refer to the next line of code. Therefore, in a script such as ours, a line directive winds up as the first line of the executable version, rendering it non-executable. The fix for this is to delete the first line directive, or move it to below the first line and increment the line number by one.

One can write filters for use with either `notangle` or `noweave` to manipulate the source once in the pipeline. The pipeline representation of the `nw` source file in `noweb` is beyond the scope of this article (see the "Noweb Hacker's Guide" included in the documentation of the distribution). We will only mention that a filter could easily be constructed which automates the solution to the above mentioned line directive problem.

The typeset version of the program is generated with the `noweave` tool. There are several useful options for `noweave`, all detailed in the man-pages. Here we will only consider a few of the most important options.

The first options of general interest concern the desired output: you may specify `-latex` (default), `-tex` or `-html` as the formatting language to be used for the final documentation. Each of these options will supply an appropriate wrapper (which can be suppressed with the `-n` option) for the typeset version. You may write your `nw` source file intended for LaTeXtypesetting and still have the option of producing an html document by invoking `noweave` with the `-html` option and the latex-to-html filter (`-filter l2h`) included with the distribution.

The `-x` option enables cross-referencing and indexing of chunk names and any identifiers which are automatically recognized by an 'autodefs' filter. Using the `-index` option implies `-x` and also provides cross-referencing and indexing for manually defined identifiers—those mentioned in `@ %def` statements in the `nw` source file.

Normally, `noweave` will insert additional information such as the filename for use in page headers with its wrapper. The `-delay` option causes `noweave` to suspend the insertion of this information until after the first documentation chunk. This is most useful when you wish to provide your own (La)TeX wrapper to specify additional packages or defining your own special formatting commands. This implies a `-n` (omission of wrapper) option and requires that you make sure to include a '
`enddocument`' control sequence in a documentation chunk at the end of the file to complete the wrapper. Our example `nw` source file is written in this fashion.

Our typeset version (Figure 1) was produced by first extracting the autodefs.perl root chunk with `notangle` and making it executable with the `chmod` system command. We then placed this executable in the `noweb` library directory and invoked `noweave` as:

```
noweave -autodefs perl -delay -index autodefs.perl.nw > autodefs.tex
```

This was followed by running LaTeXon the resulting file—twice to resolve page references—to create the dvi file, and then using dvips to create the postscript version for inclusion with this article.

Additional options allow you to have the index created from an external file, expand tabs, and to specify alternative formatting options provided by the included noweb.sty file. The latter includes options to omit chunk numbering in the left margins, change text size in code chunks, and switch from using the symbolic cross-referencing of code chunks occurring at the right margin to simple footnote style cross-referencing similar in style to the 'Defines' and 'Uses' footnotes.

Conclusion

Admittedly, a literate program in general takes a little more time and effort to initially produce. However, as much of this initial effort is devoted to explaining each part of the program, the author is likely to have produced a better quality program in the end because he or she has put more thought into the program's design at each stage of the game. Additionally, by investing in the extra effort of creating a well documented program, the time saved later in maintaining and upgrading the program is considerably lessened.

In terms of documentation and explanation, the ability to describe components as they come into play in the design of the program—rather than in the order they must occur for the compiler or interpreter—is a vast improvement over traditional commented code. In addition to the benefits of improved code and easier maintanence, literate programs can also serve well as excellent teaching tools.

Availability and Notes

`noweb` was written by Norman Ramsey, and pointers to obtaining the source and binary distributions of `noweb` (among other related resources) can be found at his `noweb` homepage (http://www.cs.virginia.edu/ nr/noweb).

The current source distribution contains both `awk` and `Icon` versions of the library files necessary. The binary version is built from the `Icon` source which is recommended as the `awk` version lacks some of the default behavior of the `Icon` version. Norman Ramsey has informed us that he is no longer able to maintain and upgrade the `awk` version.

Norman Ramsey has also told us of plans for version 2.8 to include a troff back end (in addition to the TeX, LaTeX and html back ends), conditional tangling, and some pretty printing macros.