Nuweb Version 1.56
A Simple Literate Programming Tool

Preston Briggs[1]
preston@tera.com
HTML scrap generator by John D. Ramsdell
ramsdell@mitre.org
Scrap formatting by Marc W. Mengel
mengel@fnal.gov
Continuing maintenance by Simon Wright
simon@pushface.org
and Keith Harwood
Keith.Harwood@vitalmis.com

[1]This work has been supported by ARPA, through ONR grant N00014-91-J-1989.

Contents

Chapter 1

Introduction

In 1984, Knuth introduced the idea of *literate programming* and described a pair of tools to support the practise [2]. His approach was to combine Pascal code with TEX documentation to produce a new language, `WEB`, that offered programmers a superior approach to programming. He wrote several programs in `WEB`, including `weave` and `tangle`, the programs used to support literate programming. The idea was that a programmer wrote one document, the web file, that combined documentation (written in TEX [3]) with code (written in Pascal).

Running `tangle` on the web file would produce a complete Pascal program, ready for compilation by an ordinary Pascal compiler. The primary function of `tangle` is to allow the programmer to present elements of the program in any desired order, regardless of the restrictions imposed by the programming language. Thus, the programmer is free to present his program in a top-down fashion, bottom-up fashion, or whatever seems best in terms of promoting understanding and maintenance.

Running `weave` on the web file would produce a TEX file, ready to be processed by TEX. The resulting document included a variety of automatically generated indices and cross-references that made it much easier to navigate the code. Additionally, all of the code sections were automatically prettyprinted, resulting in a quite impressive document.

Knuth also wrote the programs for TEX and METAFONT entirely in `WEB`, eventually publishing them in book form [4, 5]. These are probably the largest programs ever published in a readable form.

Inspired by Knuth's example, many people have experimented with `WEB`. Some people have even built web-like tools for their own favorite combinations of programming language and typesetting language. For example, `CWEB`, Knuth's current system of choice, works with a combination of C (or C++) and TEX [7]. Another system, FunnelWeb, is independent of any programming language and only mildly dependent on TEX [9]. Inspired by the versatility of FunnelWeb and by the daunting size of its documentation, I decided to write my own, very simple, tool for literate programming.[1]

1.1 Nuweb

Nuweb works with any programming language and LATEX [6]. I wanted to use LATEX because it supports a multi-level sectioning scheme and has facilities for drawing figures. I wanted to be able to work with arbitrary programming languages because my friends and I write programs in many languages (and sometimes combinations of several languages), *e.g.,* C, Fortran, C++, yacc, lex, Scheme, assembly, Postscript, and so forth. The need to support arbitrary programming languages has many consequences:

No prettyprinting Both `WEB` and `CWEB` are able to prettyprint the code sections of their documents because they understand the language well enough to parse it. Since we want to use *any* language, we've got to abandon this feature. However, we do allow particular individual formulas or fragments of LATEX

[1] There is another system similar to mine, written by Norman Ramsey, called *noweb* [8]. It perhaps suffers from being overly Unix-dependent and requiring several programs to use. On the other hand, its command syntax is very nice. In any case, nuweb certainly owes its name and a number of features to his inspiration.

code to be formatted and still be parts of output files. Also, keywords in scraps can be surrounded by `@_` to have them be bold in the output.

No index of identifiers Because `WEB` knows about Pascal, it is able to construct an index of all the identifiers occurring in the code sections (filtering out keywords and the standard type identifiers). Unfortunately, this isn't as easy in our case. We don't know what an identifier looks like in each language and we certainly don't know all the keywords. (On the other hand, see the end of Section 1.2.2)

Of course, we've got to have some compensation for our losses or the whole idea would be a waste. Here are the advantages I can see:

Simplicity The majority of the commands in `WEB` are concerned with control of the automatic prettyprinting. Since we don't prettyprint, many commands are eliminated. A further set of commands is subsumed by LaTeX and may also be eliminated. As a result, our set of commands is reduced to only four members (explained in the next section). This simplicity is also reflected in the size of this tool, which is quite a bit smaller than the tools used with other approaches.

No prettyprinting Everyone disagrees about how their code should look, so automatic formatting annoys many people. One approach is to provide ways to control the formatting. Our approach is simpler—we perform no automatic formatting and therefore allow the programmer complete control of code layout. We do allow individual scraps to be presented in either verbatim, math, or paragraph mode in the TeX output.

Control We also offer the programmer complete control of the layout of his output files (the files generated during tangling). Of course, this is essential for languages that are sensitive to layout; but it is also important in many practical situations, *e.g.,* debugging.

Speed Since nuweb doesn't do too much, the nuweb tool runs quickly. I combine the functions of `tangle` and `weave` into a single program that performs both functions at once.

Page numbers Inspired by the example of noweb, nuweb refers to all scraps by page number to simplify navigation. If there are multiple scraps on a page (say, page 17), they are distinguished by lower-case letters (*e.g.,* 17a, 17b, and so forth).

Multiple file output The programmer may specify more than one output file in a single nuweb file. This is required when constructing programs in a combination of languages (say, Fortran and C). It's also an advantage when constructing very large programs that would require a lot of compile time.

This last point is very important. By allowing the creation of multiple output files, we avoid the need for monolithic programs. Thus we support the creation of very large programs by groups of people.

A further reduction in compilation time is achieved by first writing each output file to a temporary location, then comparing the temporary file with the old version of the file. If there is no difference, the temporary file can be deleted. If the files differ, the old version is deleted and the temporary file renamed. This approach works well in combination with `make` (or similar tools), since `make` will avoid recompiling untouched output files.

1.1.1 Nuweb and HTML

In addition to producing LaTeX source, nuweb can be used to generate HyperText Markup Language (HTML), the markup language used by the World Wide Web. HTML provides hypertext links. When an HTML document is viewed online, a user can navigate within the document by activating the links. The tools which generate HTML automatically produce hypertext links from a nuweb source.

(Note that hyperlinks can be included in LaTeX using the `hyperref` package. This is now the preferred way of doing this and the HTML processing is not up to date.)

1.2 Writing Nuweb

The bulk of a nuweb file will be ordinary LaTeX. In fact, any LaTeX file can serve as input to nuweb and will be simply copied through, unchanged, to the documentation file—unless a nuweb command is discovered. All nuweb commands begin with an "at-sign" (`@`). Therefore, a file without at-signs will be copied unchanged. Nuweb commands are used to specify *output files,* define *fragments,* and delimit *scraps.* These are the basic features of interest to the nuweb tool—all else is simply text to be copied to the documentation file.

1.2.1 The Major Commands

Files and fragments are defined with the following commands:

`@o` ***file-name flags scrap*** Output a file. The file name is terminated by whitespace.

`@d` ***fragment-name scrap*** Define a fragment. The fragment name is terminated by a return or the beginning of a scrap.

`@q` ***fragment-name scrap*** Define a fragment. The fragment name is terminated by a return or the beginning of a scrap. This a quoted fragment.

A specific file may be specified several times, with each definition being written out, one after the other, in the order they appear. The definitions of fragments may be similarly specified piecemeal.

A fragment name may have embedded parameters. The parameters are denoted by the sequence `@'value@'` where `value` is an uninterpreted string of characters (although the sequence `@@` denotes a single `@` character). When a fragment name is used inside a scrap the parameters may be replaced by an argument which may be a different literal string, a fragment use, an embedded fragment or by a parameter use.

The difference between a quoted fragment (`@q`) and an ordinary one (`@d`) is that inside a quoted fragment fragments are not expanded on output. Rather, they are formatted as uses of fragments so that the output file can itself be `nuweb` source. This allows you to create files containing fragments which can undergo further processing before the fragments are expanded, while also describing and documenting them in one place.

You can have both quoted and unquoted fragments with the same name. They are written out in order as usual, with those introduced by `@q` being quoted and those with `@d` expanded as normal.

In quoted fragments, the `@f` filename is quoted as well, so that when it is expanded it refers to the finally produced file, not any of the intermediate ones.

Scraps

Scraps have specific begin markers and end markers to allow precise control over the contents and layout. Note that any amount of whitespace (including carriage returns) may appear between a name and the beginning of a scrap. Scraps may also appear in the running text where they are formatted (almost) identically to their use in definitions, but don't appear in any code output.

`@{`*anything*`@}` where the scrap body includes every character in *anything*—all the blanks, all the tabs, all the carriage returns. This scrap will be typeset in verbatim mode. Using the `-l` option will cause the program to typeset the scrap with the help of LaTeX's `listings` package.

`@[`*anything*`@ ]` where the scrap body includes every character in *anything*—all the blanks, all the tabs, all the carriage returns. This scrap will be typeset in paragraph mode, allowing sections of TeX documents to be scraps, but still be prettyprinted in the document.

`@(`*anything*`@)` where the scrap body includes every character in *anything*—all the blanks, all the tabs, all the carriage returns. This scrap will be typeset in math mode. This allows this scrap to contain a formula which will be typeset nicely.

Inside a scrap, we may invoke a fragment.

`@<`*fragment-name*`@>` This is a fragment use. It causes the fragment *fragment-name* to be expanded inline as the code is written out to a file. It is an error to specify recursive fragment invocations.

`@<`*fragment-name*`@( `*a1*` @, `*a2*` @) @>` This is the old form of parameterising a fragment. It causes the fragment *fragment-name* to be expanded inline with the arguments *a1*, *a2*, etc. Up to 9 arguments may be given.

`@1, @2, ..., @9` In a fragment causes the n'th fragment argument to be substituted into the scrap. If the argument is not passed, a null string is substituted. Arguments can be passed in two ways, either embedded in the fragment name or as the old form given above.

An embedded argument may specified in four ways.

`@'string@'` The string will be copied literally into the called fragment. It will not be interpreted (except for `@@` converted to `@`).

`@<`*fragment-name*`@>` The fragment will be expanded in the usual fashion and the results passed to the called fragment.

`@{text@}` The text will be expanded as normal and the results passed to the called fragment. This behaves like an anonymous fragment which has the same arguments as the calling fragment. Its principle use is to combine text and arguments into one argument.

`@1, @2, ..., @9` The argument of the calling fragment will be passed to the called fragment and expanded in there.

If an argument is used but there is no corresponding parameter in the fragment name, the null string is substituted. But what happens if there is a parameter in the full name of a fragment, but a particular application of the fragment is abbreviated (using the `. . .` notation) and the argument is missed? In that case the argument is replaced by the string given in the definition of the fragment.

In the old form the parameter may contain any text and will be expanded as a normal scrap. The two forms of parameter passing don't play nicely together. If a scrap passes both embedded and old form arguments the old form arguments are ignored.

`@x`*label*`@x` Marks a place inside a scrap and associates it to the label (which can be any text not containing a @). Expands to the reference number of the scrap followed by a numeric value. Outside scraps it expands to the same value. It is used so that text outside scraps can refer to particular places within scraps.

`@f` Inside a scrap this is replaced by the name of the current output file.

`@t` Inside a scrap this is replaced by the title of the fragment as it is at the point it is used, with all parameters replaced by actual arguments.

`@#` At the beginning of a line in a scrap this will suppress the normal indentation for that line. Use this, for example, when you have a `#ifdef` inside a nested scrap. Writing `@##ifdef` will cause it to be lined up on the left rather than indented with the rest of its code.

Note that fragment names may be abbreviated, either during invocation or definition. For example, it would be very tedious to have to type, repeatedly, the fragment name

```
@d Check for terminating at-sequence and return name if found
```

Therefore, we provide a mechanism (stolen from Knuth) of indicating abbreviated names.

```
@d Check for terminating...
```

Basically, the programmer need only type enough characters to identify the fragment name uniquely, followed by three periods. An abbreviation may even occur before the full version; nuweb simply preserves the longest version of a fragment name. Note also that blanks and tabs are insignificant within a fragment name; each string of them is replaced by a single blank.

Sometimes, for instance during program testing, it is convenient to comment out a few lines of code. In C or Fortran placing `/* ... */` around the relevant code is not a robust solution, as the code itself may contain comments. Nuweb provides the command

`@%`

only to be used inside scraps. It behaves exactly the same as `%` in the normal LaTeX text body.

When scraps are written to a program file or a documentation file, tabs are expanded into spaces by default. Currently, I assume tab stops are set every eight characters. Furthermore, when a fragment is expanded in a scrap, the body of the fragment is indented to match the indentation of the fragment invocation. Therefore, care must be taken with languages (*e.g.,* Fortran) that are sensitive to indentation. These default behaviors may be changed for each output file (see below).

Flags

When defining an output file, the programmer has the option of using flags to control output of a particular file. The flags are intended to make life a little easier for programmers using certain languages. They introduce little language dependences; however, they do so only for a particular file. Thus it is still easy to mix languages within a single document. There are four "per-file" flags:

`-d` Forces the creation of `#line` directives in the output file. These are useful with C (and sometimes C++ and Fortran) on many Unix systems since they cause the compiler's error messages to refer to the web file rather than to the output file. Similarly, they allow source debugging in terms of the web file.

`-i` Suppresses the indentation of fragments. That is, when a fragment is expanded within a scrap, it will *not* be indented to match the indentation of the fragment invocation. This flag would seem most useful for Fortran programmers.

`-t` Suppresses expansion of tabs in the output file. This feature seems important when generating `make` files.

`-c`*x* Puts comments in generated code documenting the fragment that generated that code. The *x* selects the comment style for the language in use. So far the only valid values are `c` to get `C` comment style, `+` for `C++` and `p` for `Perl`. (`Perl` commenting can be used for several languages including `sh` and, mostly, `tcl`.) If the global `-x` cross-reference flag is set the comment includes the page reference for the first scrap that generated the code.

1.2.2 The Minor Commands

We have some very low-level utility commands that may appear anywhere in the web file.

`@@` Causes a single "at sign" to be copied into the output.

`@_` Causes the text between it and the next `@_` to be made bold (for keywords, etc.)

`@i` *file-name* Includes a file. Includes may be nested, though there is currently a limit of 10 levels. The file name should be complete (no extension will be appended) and should be terminated by a carriage return. Normally the current directory is searched for the file to be included, but this can be varied using the `-I` flag on the command line. Each such flag adds one directory tothe search path and they are searched in the order given.

`@r`*x* Changes the escape character '@' to '*x*'. This must appear before any scrap definitions.

`@v` Always replaced by the string established by the `-V` flag, or a default string if the flag isn't given. This is intended to mark versions of the generated files.

Finally, there are three commands used to create indices to the fragment names, file definitions, and user-specified identifiers.

`@f` Create an index of file names.

`@m` Create an index of fragment names.

`@u` Create an index of user-specified identifiers.

I usually put these in their own section in the LaTeX document; for example, see Chapter ??.

Identifiers must be explicitly specified for inclusion in the `@u` index. By convention, each identifier is marked at the point of its definition; all references to each identifier (inside scraps) will be discovered automatically. To "mark" an identifier for inclusion in the index, we must mention it at the end of a scrap. For example,

```
@d a scrap @{
Let's pretend we're declaring the variables FOO and BAR
inside this scrap.
@| FOO BAR @}
```

I've used alphabetic identifiers in this example, but any string of characters (not including whitespace or `@` characters) will do. Therefore, it's possible to add index entries for things like `<<=` if desired. An identifier may be declared in more than one scrap.

In the generated index, each identifier appears with a list of all the scraps using and defining it, where the defining scraps are distinguished by underlining. Note that the identifier doesn't actually have to appear in the defining scrap; it just has to be in the list of definitions at the end of a scrap.

1.3 Sectioning commands

For larger documents the indexes and usage lists get rather unwieldy and problems arise in naming things so that different things in different parts of the document don't get confused. We have a sectioning command which keeps the fragment names and user identifiers separate. Thus, you can give a fragment in one section the same name as a fragment in another and the two won't be confused or connected in any way. Nor will user identifiers defined in one section be referenced in another. Except for the fact that scraps in successive sections can go into the same output file, this is the same as if the sections came from separate input files.

However, occasionally you may really want fragments from one section to be used in another. More often, you will want to identify a user identifier in one section with the same identifier in another (as, for example, a header file defined in one section is included in code in another). Extra commands allow a fragment defined in one section to be accessible from all other sections. Similarly, you can have scraps which define user identifiers and export them so that they can be used in other sections.

`@s` Start a new section.

`@S` Close the current section and don't start another.

`@d+` **fragment-name scrap** Define a fragment which is accessible in all sections, a global fragment.

`@D+` Likewise

`@q+` Likewise

`@Q+` Likewise

`@m+` Create an index of all such fragments.

`@u+` Create an index of globally accessible user identifiers.

There are two kinds of section, the base section which is where normally everything goes, and local sections which are introduced with the `@s` command. A local section comprises everything from the command which starts it to the one which ends it. A `@s` command will start a new local section. A `@S` command closes the current local section, but doesn't open another, so what follows goes into the base section. Note that fragments defined in the base section aren't global; they are accessible only in the base section, but they are accessible regardless of any local sections between their definition and their use.

Within a scrap:

`@<+`*fragment-name*`@>` Expand the globally accessible fragment with that name, rather than any local fragment.

`@+` Like `@|` except that the identifiers defined are exported to the global realm and are not directly referenced in any scrap in any section (not even the one where they are defined).

`@-` Like `@|` except that the identifiers are imported to the local realm. The cross-references show where the global variables are defined and defines the same names as locally accesible. Uses of the names within the section will point to this scrap.

Note that the `+` signs above are part of the commands. They are not part of the fragment names. If you want a fragment whose name begins with a plus sign, leave a space between the command and the name.

1.4 Running Nuweb

Nuweb is invoked using the following command:

`nuweb` *flags file-name...*

One or more files may be processed at a time. If a file name has no extension, `.w` will be appended. LaTeX suitable for translation into HTML by LaTeX2HTML will be produced from files whose name ends with `.hw`, otherwise, ordinary LaTeX will be produced. While a file name may specify a file in another directory, the resulting documentation file will always be created in the current directory. For example,

```
nuweb /foo/bar/quux
```

will take as input the file `/foo/bar/quux.w` and will create the file `quux.tex` in the current directory.

By default, nuweb performs both tangling and weaving at the same time. Normally, this is not a bottleneck in the compilation process; however, it's possible to achieve slightly faster throughput by avoiding one or another of the default functions using command-line flags. There are currently three possible flags:

`-t` Suppress generation of the documentation file.

`-o` Suppress generation of the output files.

`-c` Avoid testing output files for change before updating them.

Thus, the command

```
nuweb -to /foo/bar/quux
```

would simply scan the input and produce no output at all.

There are several additional command-line flags:

`-v` For "verbose", causes nuweb to write information about its progress to `stderr`.

`-n` Forces scraps to be numbered sequentially from 1 (instead of using page numbers). This form is perhaps more desirable for small webs.

`-s` Doesn't print list of scraps making up each file following each scrap.

`-d` Print "dangling" identifiers – user identifiers which are never referenced, in indices, etc.

`-p` *path* Prepend *path* to the filenames for all the output files.

`-l` Use the `listings` package for formatting scraps. Use this if you want to have a pretty-printer for your scraps. In order to e.g. have pretty Perl scraps, include the following LaTeX commands in your document:

```
\usepackage{listings}
...
\lstset{extendedchars=true,keepspaces=true,language=perl}
```

See the `listings` documentation for a list of formatting options. Be sure to include a `\usepackage{listings}` in your document.

`-V` *string* Provide *string* as the replacement for the @v operation.

1.5 Generating HTML

Nikos Drakos' LaTeX2HTML Version 0.5.3 [1] can be used to translate LaTeX with embedded HTML scraps into HTML. Be sure to include the document-style option `html` so that LaTeX will understand the hypertext commands. When translating into HTML, do not allow a document to be split by specifying "`-split 0`". You need not generate navigation links, so also specify "`-no_navigation`".

While preparing a web, you may want to view the program's scraps without taking the time to run LaTeX2HTML. Simply rename the generated LaTeX source so that its file name ends with `.html`, and view that file. The documentations section will be jumbled, but the scraps will be clear.

(Note that the HTML generation is not currently maintained. If the only reason you want HTML is ti get hyperlinks, use the LaTeX `hyperref` package and produce your document as PDF via `pdflatex`.)

1.6 Restrictions

Because nuweb is intended to be a simple tool, I've established a few restrictions. Over time, some of these may be eliminated; others seem fundamental.

- The handling of errors is not completely ideal. In some cases, I simply warn of a problem and continue; in other cases I halt immediately. This behavior should be regularized.
- I warn about references to fragments that haven't been defined, but don't halt. The name of the fragment is included in the output file surrounded by <> signs. This seems most convenient for development, but may change in the future.
- File names and index entries should not contain any `@` signs.
- Fragment names may be (almost) any well-formed TeX string. It makes sense to change fonts or use math mode; however, care should be taken to ensure matching braces, brackets, and dollar signs. When producing HTML, fragments are displayed in a preformatted element (PRE), so fragments may contain one or more A, B, I, U, or P elements or data characters.
- Anything is allowed in the body of a scrap; however, very long scraps (horizontally or vertically) may not typeset well.
- Temporary files (created for comparison to the eventual output files) are placed in the current directory. Since they may be renamed to an output file name, all the output files should be on the same file system as the current directory. Alternatively, you can use the `-p` flag to specify where the files go.
- Because page numbers cannot be determined until the document has been typeset, we have to rerun nuweb after LaTeX to obtain a clean version of the document (very similar to the way we sometimes have to rerun LaTeX to obtain an up-to-date table of contents after significant edits). Nuweb will warn (in most cases) when this needs to be done; in the remaining cases, LaTeX will warn that labels may have changed.

Very long scraps may be allowed to break across a page if declared with `@O` or `@D` (instead of `@o` and `@d`). This doesn't work very well as a default, since far too many short scraps will be broken across pages; however, as a user-controlled option, it seems very useful. No distinction is made between the upper case and lower case forms of these commands when generating HTML.

1.7 Acknowledgements

Several people have contributed their times, ideas, and debugging skills. In particular, I'd like to acknowledge the contributions of Osman Buyukisik, Manuel Carriba, Adrian Clarke, Tim Harvey, Michael Lewis, Walter Ravenek, Rob Shillingsburg, Kayvan Sylvan, Dominique de Waleffe, and Scott Warren. Of course, most of these people would never have heard or nuweb (or many other tools) without the efforts of George Greenwade.

Since maintenance has been taken over by Marc Mengel, Simon Wright and Keith Harwood online contributions have been made by:

- Walter Brown `<wb@fnal.gov>`
- Nicky van Foreest `<n.d.vanforeest@math.utwente.nl>`
- Javier Goizueta `<jgoizueta@jazzfree.com>`
- Alan Karp `<karp@hp.com>`

Bibliography

[1] Nikos Drakos, *From text to hypertext: A post-hoc rationalisation of latex2html*, Computer Networks and ISDN Systems **27** (1994), 215–224. 8

[2] Donald E. Knuth, *Literate programming*, The Computer Journal **27** (1984), no. 2, 97–111. 1

[3] ———, *The TEXbook*, Computers and Typesetting, vol. 1986aA, Addison-Wesley, Reading, MA, USA, 1986. 1

[4] ———, *TEX: The program*, Computers and Typesetting, vol. B, Addison-Wesley, Reading, MA, USA, 1986b1986. 1

[5] ———, METAFONT: THE PROGRAM, Computers and Typesetting, vol. D, Addison-Wesley, Reading, MA, USA, 1986d1986. 1

[6] Leslie Lamport, *LATEXa document preparation systemuser's guide and reference manual*, Addison-Wesley, Reading, MA, USA, 1985. 1

[7] Silvio Levy, *WEB adapted to C, another approach*, TUB **8** (1987), no. 1, 12–13. 1

[8] Norman Ramsey, *Literate-programming tools need not be complex*, Report at `ftp.cs.princeton.edu` in `/reports/1991/351.ps.Z`. Software at `ftp.cs.princeton.edu` in `/pub/noweb.shar.Z` and at `bellcore.com` in `/pub/norman/noweb.shar.Z`. CS-TR-351-91, Department of Computer Science, Princeton University, August 1992, Submitted to *IEEE Software*. 1

[9] Ross Williams, *Funnelweb user's manual*, `ftp.adelaide.edu.au` in `/pub/compression` and `/pub/funnelweb`, University of Adelaide, Adelaide, South Australia, Australia, 1992. 1