The noweb Hacker's Guide

Norman Ramsey*
Department of Computer Science
Princeton University

September 1992
(Revised August 1994, December 1997)

Abstract

Noweb is unique among literate-programming tools in its pipelined architecture, which makes it easy for users to change its behavior or to add new features, without even recompiling. This guide describes the representation used in the pipeline and the behavior of the existing pipeline stages. Ordinary users will find nothing of interest here; the guide is addressed to those who want to change or extend noweb.

*Author's current address is Division of Engineering and Applied Sciences, Harvard University, 33 Oxford St, Cambridge, MA 02138, USA; send email to nr@eecs.harvard.edu.

Contents

List of Tables

Introduction

Ramsey (1994) describes `noweb` from a user's point of view, showing its simplicity and examples of its use. The `noweb` tools are implemented as *pipelines.* Each pipeline begins with the `noweb` source file. Successive stages of the pipeline implement simple transformations of the source, until the desired result emerges from the end of the pipeline. Figures 1 and 2 on page 16 show pipelines for `notangle` and `noweave`. Pipelines are responsible for `noweb`'s extensibility, which enables its users to create new literate-programming features without having to write their own tools. This document explains how to change or extend `noweb` by inserting or removing pipeline stages. Readers should be familiar with the `noweb` man pages, which describe the structure of `noweb` source files.

`Markup`, which is the first stage in every pipeline, converts `noweb` source to a representation easily manipulated by common Unix tools like `sed` and `awk`, simplifying the construction of later pipeline stages. Middle stages add information to the representation. `notangle`'s final stage converts to code; `noweave`'s final stages convert to TeX, LaTeX or HTML. Middle stages are called *filters*, by analogy with Unix filters. Final stages are called *back ends*, by analogy with compilers—they don't transform `noweb`'s intermediate representation; they emit something else.

The pipeline representation

In the pipeline, every line begins with an at sign and one of the keywords shown in Table 1. The structural keywords represent the `noweb` source syntax directly. They must appear in particular orders that reflect the structure of the source. The tagging keywords can be inserted essentially anywhere (within reason), and with some exceptions, they are not generated by `markup`. The wrapper keywords mark the beginning and end of file, and they carry information about what formatters are supposed to do in the way of leading and trailing boilerplate. They are used by `noweave` but not by `notangle`, and they are inserted directly by the `noweave` shell script, not by `markup`.

Structural keywords

The structural keywords represent the chunks in the `noweb` source. Each chunk is bracketed by a `@begin`... `@end` pair, and the *kind* of chunk is either `docs` or `code`. The `@begin` and `@end` are numbered; within a single file, numbers must be monotonically increasing, but they need not be consecutive. Filters may change chunk numbers at will.

Depending on its kind, a chunk may contain *documentation* or *code*. Documentation may contain text and newlines, represented by `@text` and `@nl`. It may also contain *quoted code* bracketed by `@quote`... `@endquote`. Every `@quote` must be terminated by an `@endquote` within the same chunk. Quoted code corresponds to the `[[...]]` construct in the `noweb` source.

Structural keywords

`@begin` *kind n*	Start a chunk
`@end` *kind n*	End a chunk
`@text` *string*	*string* appeared in a chunk
`@nl`	A newline appeared in a chunk
`@defn` *name*	The code chunk named *name* is being defined
`@use` *name*	A reference to code chunk named *name*
`@quote`	Start of quoted code in a documentation chunk
`@endquote`	End of quoted code in a documentation chunk

Tagging keywords

`@file` *filename*	Name of the file from which the chunks came
`@line` *n*	Next text line came from source line *n* in current file
`@language` *language*	Programming language in which code is written
`@index ...`	Index information.
`@xref ...`	Cross-reference information.

Wrapper keywords

`@header` *formatter options*	First line, identifying formatter and options
`@trailer` *formatter*	Last line, identifying formatter.

Error keyword

`@fatal` *stagename message*	A fatal error has occurred.

Lying, cheating, stealing keyword

`@literal` *text*	Copy *text* to output.

Table 1: Keywords used in `noweb`'s pipeline representation

Code, whether it appears in quoted code or in a code chunk, may contain text and newlines, and also definitions and uses of code chunks, marked with `@defn` and `@use`. The first structural keyword in any code chunk must be `@defn`. `@defn` may be preceded or followed by tagging keywords, but the next structural keyword must be `@nl`; together, the `@defn` and `@nl` represent the initial `<<chunk name>>=` that starts the chunk (including the terminating newline).

A few facts follow from what's already stated above, but are probably worth noting explicitly:

- Quoted code may not appear in code, nor may it appear in `@defn` or `@use`. `noweave` back ends are encouraged to give `[[...]]` special treatment when it appears in `defn` or `use`, so that the text contained therein is treated as if it were quoted code.

- The text in chunks may be distributed among as many `@text` keywords as desirable. Any number of empty `@text` keywords are permitted. In particular, it is not realistic to expect that a single line will be represented in a single `@text` (see the discussion of `finduses` on page 14).

- `markup` will sometimes emit `@use` within `@quote`... `@endquote`, for example from a source like `[[<<chunk name>>]]`.

- No two chunks have the same number.

- Because later filters can change chunk numbers, no filter should plant references to chunk numbers anywhere in the pipeline.

Tagging keywords

The structural keywords carry all the code and documentation that appears in a `noweb` source file. The tagging keywords carry information about that code or documentation. The `@file` keyword carries the name of the source file from which the following lines come. The `@line` keyword give the line number of the next `@text` line within the current file (as determined by the most recent `@file` keyword). The only guarantee about where these appear is that `markup` introduces each new source file with a `@file` that appears between chunks. Most filters ignore `@file` and `@line`, but `nt` respects them, so that `notangle` can properly mark line numbers if some `noweb` filter starts moving lines around.

Programming languages

To support automatic indexing or prettyprinting, it's possible to indicate the programming language in which a chunk is written. The `@language` keyword may appear at most once between each `@begin code` and `@end code` pair. Standard values of `@language` and their associated meanings are:

`awk`	awk
`c`	C
`c++`	C++
`caml`	CAML
`html`	HTML
`icon`	Icon
`latex`	LaTeX source
`lisp`	Lisp or Scheme
`make`	A Makefile
`m3`	Modula-3
`ocaml`	Objective CAML
`perl`	A perl script
`python`	Python
`sh`	A shell script
`sml`	Standard ML
`tex`	plain TeX
`tcl`	tcl

If the `@language` keyword catches on, it may be useful to create an automatic registry on the World-Wide Web.

I have made it impossible to place `@language` information directly in a `noweb` source file. My intent is that tools will identify the language of the root chunks using any of several methods: conventional names of chunks, being told on a command line, or identifying the language by looking at the content of the chunks. (Of these methods, the most practical is to name the root chunks after the files to which they will be extracted, and to use the same naming conventions as `make` to figure out what the contents are.) A `noweb` filter will tag non-root chunks with the appropriate `@language` by propagating information from uses to definitions.

Indexing and cross-reference concepts

The index and cross-reference commands use *label*s, *ident*s, and *tag*s. A *label* is a unique string generated to refer to some element of a literate program. They serve as labels or "anchor points" for back ends that are capable of implementing their own cross-reference. So, for example, the LaTeX back end uses labels as arguments to `\label` and `\ref`, and the HTML back end uses labels to name and refer to anchors. Labels never contain white space, which simplifies parsing. The standard filters cross-reference at the chunk level, so that each label refers to a particular code chunk, and all references to that chunk use the same label.

An *ident* refers to a source-language identifier. `Noweb`'s concept of identifier is general; an identifier is an arbitrary string. It can even contain whitespace. Identifiers are used as keys in the index; references to the same string are assumed to denote the same identifier.

*Tag*s are the strings used to identify components for cross-reference in the final document. For example, Classic `WEB` uses consecutive "section numbers" to refer to chunks. `Noweb`, by default, uses "sub-page references," e.g., "24b" for the second chunk appearing on page 24. The HTML back end doesn't use any tags at all; instead, it implements cross-referencing using the "hot link" mechanism.

The final step of cross-referencing involves generating tags and associating a tag with each label. All the existing back ends rely on a document formatter to do this job, but that strategy might be worth changing. Computing tags within a `noweb` filter could be lots easier than doing it in a formatter. For example, a filter that computed sub-page numbers by grubbing in `.aux` files would be pretty easy to write, and it would eliminate a lot of squirrely LaTeX code.

Index information

I've divided the index keywords into several groups. There seems to be a plethora of keywords, but most of them are straightforward representations of parts of a document produced by `noweave`. Readers may want to have a sample of `noweave`'s output handy when studying this and the next section.

Definitions, uses, and `@ %def` `@index defn`, `@index use`, and `@index nl` are the only `@index` keywords that appear in `markup`'s output, and thus which can appear in any program. They may appear only within the boundaries of a code chunk (`@begin code`... `@end code`). `@index defn` and `@index use` simply indicate that the current chunk contains a definition or use of the identifier *ident* which follows the keyword. The placement of `@index defn` need not bear a relationship to the text of the definition, but `@index use` is normally followed by a `@text` that contains the source-code text identified as the use.[1]

Instances of `@index defn` normally come from one of two sources: either a language-dependent recognizer of definitions, or a hand-written `@ %def` line.[2] In the latter case, the line is terminated by a newline that is neither part of a code chunk nor part of a documentation chunk. To keep line numbers accurate, that newline can't just be abandoned, but neither can it be represented by `@nl` in a documentation or code chunk. The solution is the `@index nl` keyword, which serves no purpose other than to keep track of these newlines, so that back ends can produce accurate line numbers.

Following a suggestion by Oren Ben-Kiki, `@index localdefn` indicates a definition that is not to be visible outside the current file. It may be produced by a language-dependent recognizer or other filter. Because I have questions about the need for `@index localdefn`, there is officially no way to cause `markup` to produce it.

[1]This property can't hold when one identifier is a prefix of another; see the description of `finduses` on page 14.

[2]The `@ %def` notation has been deprecated since version 2.10.

Definitions, uses, and `@ %def`

`@index defn` *ident*	The current chunk contains a definition of *ident*
`@index localdefn` *ident*	The current chunk contains a definition of *ident*, which is not to be visible outside this file
`@index use` *ident*	The current chunk contains a use of *ident*
`@index nl` *ident*	A newline that is part of markup, not part of the chunk

Identifiers defined in a chunk

`@index begindefs`	Start list of identifiers defined in this chunk
`@index isused` *label*	The identifier named in the following `@index defitem` is used in the chunk labelled by *label*
`@index defitem` *ident*	*ident* is defined in this chunk, and it is used in all the chunks named in the immediately preceding `@index isused`.
`@index enddefs`	End list of identifiers defined in this chunk

Identifiers used in a chunk

`@index beginuses`	Start list of identifiers used in this chunk
`@index isdefined` *label*	The identifier named in the following `@index useitem` is defined in the chunk labelled by *label*
`@index useitem` *ident*	*ident* is used in this chunk, and it is defined in each of the chunks named in the immediately preceding `@index isdefined`.
`@index enduses`	End list of identifiers used in this chunk

The index of identifiers

`@index beginindex`	Start of the index of identifiers
`@index entrybegin` *label* *ident*	Beginning of the entry for *ident*, whose first definition is found at *label*
`@index entryuse` *label*	A use of the identifer named in the last `@index entrybegin` occurs at the chunk labelled with *label*.
`@index entrydefn` *label*	A definition of the identifer named in the last `@index entrybegin` occurs at the chunk labelled with *label*.
`@index entryend`	End of the entry started by the last `@index entrybegin`
`@index endindex`	End of the index of identifiers

Table 2: Indexing keywords

Identifiers defined in a chunk The keywords from `@index begindefs` to `@index enddefs` are used to represent a more complex data structure giving the list of identifiers defined in a code chunk. The constellation represents a list of identifiers; one `@index defitem` appears for each identifier. The group also tells in what other chunks each identifier is used; those chunks are listed by `@index isused` keywords which appear just before `@index defitem`. The labels in these keywords appear in the order of the corresponding code chunks, and there are no duplicates.

These keywords can appear anywhere inside a code chunk, but filters are encouraged to keep these keywords together. The standard filters guarantee that only `@index isused` and `@index defitem` appear between `@index begindefs` and `@index enddefs`. The standard filters put them at the end of the code chunk, which simplifies translation by the LaTeX back end, but that strategy might change in the future.

It should go without saying, but the keywords in these and all similar groups (including some `@xref` groups) must be properly structured. That is to say:

1. Every `@index begindefs` must have a matching `@index enddefs` within the same code chunk.
2. `@index isused` and `@index defitem` may appear only between matching `@index begindefs` and `@index enddefs`.
3. The damn things can't be nested.

Identifiers used in a chunk The keywords from `@index beginuses` to `@index enduses` are the dual of `@index begindef` to `@index enddef`; the structure lists the identifiers used in the current code chunk, with cross-references to the definitions. Similar interpretations and restrictions apply. Note that an identifier can be defined in more than one chunk, although we expect that to be an unusual event.

The index of identifiers Keywords `@index beginindex` to `@index endindex` represent the complete index of all the identifiers used in the document. Each entry in the index is bracketed by `@index entrybegin`... `@index entryend`. An entry provides the name of the identifier, plus the labels of all the chunks in which the identifier is defined or used. The label of the first defining chunk is given at the beginning of the entry so that back ends needn't search for it.

Filters are encouraged to keep these keywords together. The standard filters put them almost at the very end of the `noweb` file, just before the optional `@trailer`.

Cross-reference information

The most basic function of the cross-referencing keywords is to associate labels and pointers (cross-references) with elements of the document, which is done with the `@xref ref` and `@xref label` keywords. The other `@xref` keywords all express chunk cross-reference information that is emitted directly by one or more back ends.

Chunk cross-reference introduces the idea of an *anchor*, which is a label that refers to an "interesting point" we identify with the beginning of a code chunk. The anchor is the place we expect to turn when we want to know about a code chunk; its exact value and interpretation depend on the back end being used. The standard LaTeX back end uses the sub-page number of the defining chunk as the anchor, but the standard HTML back end uses some `@text` from the documentation chunk preceding the code chunk.

Basic cross-reference `@xref label` and `@xref ref` are named by analogy with the LaTeX `\label` and `\ref` commands. `@xref label` is used to associate a *label* with a succeeding item. Items that can be so labelled include

`@defn`	Labels the code chunk that begins with this `@defn`.
`@use`	Labels this particular use.
`@index defn`	Labels this definition of an identifier.
`@index use`	Labels this use of an identifier.
`@text`	Typically labels part of a documentation chunk.
`@end docs`	Typically labels an empty documentation chunk.

I haven't made up my mind whether this should be the complete set, but these are the ones used by the standard filters. Most back ends use the chunk as the basic unit of cross-reference, so the labels of `@defn` are the ones that are most often used. The HTML back end, however, does something a little different—it uses labels that refer to documentation preceding a chunk, because the typical HTML browser (Mosaic) places the label[3] at the top of the screen, and using the label of the `@defn` would lose the documentation immediately preceding a chunk. The labels used by this back end usually point to `@text`, but they may point to `@end docs` when no text is available.

`@xref ref` is used to associate a reference with a succeeding item. Such items include

`@defn`, `@use`	Refers to the label used as an *anchor* for this chunk.
`@index defn`, `@index use`	Refers to the label used as an *anchor* for the first chunk in which this identifier is defined.

Linking previous and next definitions of a code chunk `@xref prevdef` and `@xref nextdef` may appear anywhere in a code chunk, and they give the labels of the preceding and succeeding definitions of that code chunk, if any. Standard filters currently put them at the beginning of the code chunk, following the initial `@defn`, so the information can be used on the `@defn` line, à la Fraser and Hanson (1995).

[3]The HTML terminology calls a label an "anchor."

Basic cross-reference

`@xref label` *label*	Associates *label* with tagged item.
`@xref ref` *label*	Cross-reference from tagged item to item associated with *label*.

Linking previous and next definitions of a code chunk

`@xref prevdef` *label*	The `@defn` from the previous definition of this chunk is associated with *label*.
`@xref nextdef` *label*	The `@defn` from the next definition of this chunk is associated with *label*.

Continued definitions of the current chunk

`@xref begindefs`	Start "This definition is continued in ..."
`@xref defitem` *label*	Gives the label of a chunk in which the definition of the current chunk is continued.
`@xref enddefs`	Ends the list of chunks where definition is continued.

Chunks where this code is used

`@xref beginuses`	Start "This code is used in ..."
`@xref useitem` *label*	Gives the label of a chunk in which this chunk is used.
`@xref enduses`	Ends the list of chunks in which this code is used.
`@xref notused` *name*	Indicates that this chunk isn't used anywhere in this document.

The list of chunks

`@xref beginchunks`	Start of the list of chunks
`@xref chunkbegin` *label name*	Beginning of the entry for chunk *name*, whose *anchor* is found at *label*.
`@xref chunkuse` *label*	The chunk is used in the chunk labelled with *label*.
`@xref chunkdefn` *label*	The chunk is defined in the chunk labelled with *label*.
`@xref chunkend`	End of the entry started by the last `@xref chunkbegin`
`@xref endchunks`	End of the list of chunks

Converting labels to tags

`@xref tag` *label tag*	Associates *label* with *tag*.

Table 3: Cross-referencing keywords

Continued definitions of the current chunk The keywords ranging from `@xref begindefs` to `@xref enddefs` appear in the first definition of each code chunk. They provide the information needed by the "This definition is continued in ..." message printed by the standard LaTeX back end. They can appear anywhere in a code chunk, but standard filters put them after all the `@text` and `@nl`s, so that back ends can just print out text.

Chunks where this code is used The keywords from `@xref beginuses` to `@xref enduses` are the dual of `@xref begindefs` to `@xref enddefs`; they show where the current chunk is used. As with `@xref begindefs`... `@xref enddefs`, they appear only in the first definition of any code chunk, and they come at the end. Sometimes, as with root chunks, the code isn't used anywhere, in which case `@xref notused` appears instead of `@xref beginuses`... `@xref enduses`. The name of the current chunk appears as an argument to `@xref notused` because some back ends may want to print a special message for unused chunks—they might be written to files, for example.

The list of chunks The list of chunks, which is defined by the keywords `@xref beginchunks`... `@xref endchunks`, is the analog of the index of identifiers, but it lists all the code chunks in the document, not all the identifiers.

Filters are encouraged to keep these keywords together. The standard filters put them at the end of the `noweb` file, just before the index of identifiers.

Converting labels to tags None of the existing back ends actually computes tags; they all use formatting engines to do the job. The LaTeX back end uses an elaborate macro package to compute sub-page numbers, and the HTML back end arranges for "hot links" to be used instead of textual tags. Some people have argued that literate-programming tools shouldn't require elaborate macro packages, that they should use the basic facilities provided by a formatter. Nuweb, for example, uses standard LaTeX commands only, but goes digging through `.aux` files to find labels and compute sub-page numbers. Doing this kind of computation in a real programming language is much easier than doing it with TeX macros, and I expect that one day `noweb` will have a tag-computing filter, the results of which will be expressed using the `@xref tag` keyword.

The rules governing `@xref tag` are that it can appear anywhere. None of the standard filters or back ends does anything with it.

Wrapper keywords

The wrapper keywords, `@header` and `@trailer`, are anomalous in that they're not generated by `markup` or by any of the standard filters; instead they're inserted by the `noweave` shell script at the very beginning and end of file. The standard TeX, LaTeX, and HTML back ends use them to provide preamble and postamble markup, i.e., boilerplate that usually has to surround a document. They're not required (sometimes you don't want that boilerplate), but when they appear they must be the very first and last lines in the file, and the formatter names must match.

Error keyword

The error keyword `@fatal` signifies that a fatal error as occurred. The pipeline stage originating such an error gives its own name and a message, and it also writes a message to standard error. Filters seeing `@fatal` must copy it to their output and terminate themselves with error status. Back ends seeing `@fatal` must terminate themselves with error status. (They should not write anything to standard error since that will have been done.)

Using `@fatal` enables shell scripts to detect that something has gone wrong even if the only exit status they have access to is the exit status of the last stage in a pipeline.

Lying, cheating, stealing keyword

The `@literal` keyword is used to hack output directly into `noweave` back ends, like `totex` and `tohtml`. These back ends simply copy the text to their output. Tangling back ends ignore `@literal`. The `@literal` keyword is used by Master Hackers who are too lazy to write new back ends. Its use is deprecated. It should not exist. But it will be retained forever in the name of Backward Compatibility.

Standard filters

All the standard filters, unless otherwise noted, read the `noweb` keyword format on standard input and write it on standard output. Some filters may also use auxiliary files.

`markup`

Strictly speaking, `markup` is a front end, not a filter, but I discuss it along with filters because it generates the output that is massaged by all the filters. `markup`'s output represents a sequence of files. Each file is represented by a "`@file` *filename*" line, followed by a sequence of chunks. `markup` numbers chunks consecutively, starting at 0. It also recognizes and undoes the escape sequence for double brackets, e.g. converting "`@<<`" to "`<<`". The only tagging keywords found in its output are `@index defn` or `@index nl`; despite what's written about it, `@index use` never appears.

`autodefs.*`

I've written half a dozen language-dependent filters that use simple heuristics ("fuzzy parsing" if you prefer) to try to identify interesting definitions of identifiers. Many of these doubtless rely on my own idiosyncratic coding styles, but all of them provide good value for little effort. None of them does anything more complicated than scan individual `@text` lines in code chunks, spitting out `@index defn` and `@index localdefn` lines after the `@text` line whenever it thinks it's found something. All the filters are written in Icon and use a central core defined in `icon/defns.nw`. The C filter is the most complicated; it actually tries to understand parts of the C grammar for declarations. None of these filters has any command-line options.

`finduses`

Using code contributed by Preston Briggs, this filter makes two passes over its input. The first pass reads in all the `@index defn` and `@index localdefn` lines and builds an Aho-Corasick recognizer for the identifiers named therein. The second pass copies the input, searching for these identifiers in each `@text` line that is code. When it finds an identifier, `finduses` breaks the `@text` line into pieces, inserting `@index use` immediately before the `@text` piece that contains the identifier just found.[4] `finduses` assumes that previous filters will not have broken `@text` lines in the middle of identifiers.

The `-noquote` command-line option prevents `finduses` from searching for uses in quoted code. If `finduses` is given arguments, it takes those arguments to be file names, and it reads lists of identifiers (one per line) from the files so named, rather than from its input. This technique enables `finduses` to make a single pass over its input; `noweave` uses it to implement the `-indexfrom` option.

`finduses` shouldn't be run before filters which, like the `autodefs` filters, expect one line to be represented in a single `@text`. Filters (or back ends) that have to be run late, like prettyprinters, should be prepared to deal with lines broken into pieces and with `@index` and `@xref` tags intercalated.

`noidx`

`noidx` computes all the index and cross-reference information represented by the `@index` and `@xref` keywords.

The `-delay` command-line option delays heading material until after the first chunk, and brings trailing material before the last chunk. In particular, it causes the list of chunks and the index of identifiers to be emitted before the last chunk.

The `-docanchor` n option sets the anchor for a code chunk to be either:

1. If a documentation chunk precedes the code chunk and is n or more lines long, n lines from the end of that documentation chunk.

2. If a documentation chunk precedes the code chunk and is fewer than n lines long, at the beginning of that documentation chunk.

[4] The behavior described would duplicate `@text` pieces whenever one identifier was a prefix of another. This event is rare, and probably undesirable, but it can happen if, for example, the C++ names `MyClass` and `MyClass::Function` are both considered identifiers. In this case, whatever identifier is found first is emitted first, and only the unemitted pieces of longer identifiers are emitted.

3. If no documentation chunk precedes the code chunk, at the beginning of the code chunk, just as if `-docanchor` had not been used.

This option is used to create anchors suitable for the HTML back end.

Standard back ends

`nt`

The `nt` back end implements `notangle`. It extracts the program defined by a single code chunk (expanding all uses to form their definitions) and writes that program on standard output. Its command-line options are:

`-t`	Turn off expansion of tabs.
`-t`*n*	Expand tabs on *n*-column boundaries.
`-R`*name*	Expand the code chunk named *name*.
`-L`*format*	Use *format* as the format string to emit line-number information.

See the man page for `notangle` for details on the operation of `nt`.

`mnt`

`mnt` (for Multiple NoTangle) is a back end that can extract several code chunks from a single document in a single pass. It is used to make the `noweb` shell script more efficient. In addition to the `-t` and `-L` options recognized by `nt`, it recognizes `-all` as an instruction to extract and write to files all of the code chunks that conform to the rules set out in the `noweb` man page. It also accepts arguments, as well as options; each argument is taken to be the name of a code chunk that should be emitted to the file of the same name. Unlike `nt`, `mnt` has the function of `cpif` built in—it writes to a temporary file, then overwrites an existing file only if the temporary file is different.

`tohtml`

This back end emits HTML. It uses the formatter `html` with `@header` and `@trailer` to emit suitable HTML boilerplate. For other formatters (like `none`) it emits no header or trailer. Its command-line options are:

`-delay`	Accepted, for compatibility with other back ends, but ignored.
`-localindex`	Produces local identifier cross-reference after each code chunk.
`-raw`	Wraps text generated for code chunks in a LaTeX `rawhtml` environment, making the whole document suitable for processing with `latex2html`.

`markup`: Convert to pipeline representation
  `nt`: Extract desired chunk to standard output

Figure 1: Stages in pipeline for `notangle`

`markup`: Convert to pipeline representation
  `autodefs.c`: Find definitions in C code
    `finduses -noquote`: Find uses of defined identifiers
      `noidx`: Add index and cross-reference information
        `totex`: Convert to LaTeX

Figure 2: Stages in pipeline for `noweave -index -autodefs c`

`totex`

`totex` implements both the plain TeX and LaTeX back ends, using `@header tex` and `@header latex` to distinguish them. When using a LaTeX header, `totex` places the optional text following the header inside a `\noweboptions` command.

On the command line, the `-delay` option makes `totex` delay filename markup until after the first documentation chunk; this behavior makes the first documentation chunk a "limbo" chunk, which can usefully contain commands like `\documentclass`. The `-noindex` option suppresses output relating to the index of identifiers; it is used to implement `noweave -x`.

`unmarkup`

`unmarkup` attempts to be the inverse of markup—a document already in the pipeline is converted back to `noweb` source form. This back end is useful primarily for trying to convert other literate programs to `noweb` form. It might also be used to capture and edit the output of an automatic definition recognizer.

Standard commands

The standard commands are all written as Bourne shell scripts (Kernighan and Pike 1984). They assemble Unix pipelines using `markup` and the filters and back ends described above. They are documented in man pages, and there is no sense in repeating that material here. I do show two sample pipelines in Figures 1 and 2. The source code is available in the `shell` directory for those who want to explore further.

```
awk 'BEGIN { line = 0; capture = 0
             format = sprintf("'"$format"'",'"$width"')
           }
function comment(s) {
    '"$subst"'
    return sprintf(format,s)
}

function grab(s) {
  if (capture==0) print
  else holding[line] = holding[line] s
}

/^@end doc/ { capture = 0; holding[++line] = "" ; next }
/^@begin doc/ { capture = 1; next }

/^@text /     { grab(substr($0,7)); next}
/^@quote$/    { grab("[[") ; next}
/^@endquote$/ { grab("]]") ; next}

/^@nl$/ { if (capture !=0 ) {
            holding[++line] = ""
          } else if (defn_pending != 0) {
            print "@nl"
            for (i=0; i<=line && holding[i] ~ /^ *$/; i++) i=i
            for (; i<=line; i++)
              printf "@text %s\n@nl\n", comment(holding[i])
            line = 0; holding[0] = ""
            defn_pending = 0
          } else print
          next
        }

/^@defn / { holding[line] = holding[line] "<"substr($0,7)">="
            print ; defn_pending = 1 ; next }
{ print }'
```

Figure 3: `awk` command used to transform documentation to comments `$subst`, `$format`, and `$width` are shell variables used to adapt the script for different languages. executing `$subst` eliminates comment-end markers (if any) from the documentation, and the initial `sprintf` that creates the `awk` variable `format` gives the format used to print a line of documentation as a comment.

Examples

I don't give examples of the pipeline representation; it's best just to play with the existing filters. In particular,

`noweave -v` *options inputs* `>/dev/null`

prints (on standard error) the pipeline used by `noweave` to implement any set of *options*. In this section, I give examples of a few nonstandard filters I've thrown together for one purpose or another.

This one-line `sed` command makes `noweb` treat two chunk names as identical if they differ only in their representation of whitespace:

```
sed -e '/^@use /s/[ \t][ \t]*/ /g' -e '/^@defn /s/[ \t][ \t]*/ /g'
```

This little filter, a Bourne shell script written in `awk` (Aho, Kernighan, and Weinberger 1988), makes the definition of an empty chunk (`<<>>=`) stand for a continuation of the previous chunk definition.

```
awk 'BEGIN { lastdefn = "@defn " }
/^@defn $/ { print lastdefn; next }
/^@defn /  { lastdefn = $0 }
{ print }' "$@"
```

To share programs with colleagues who don't enjoy literate programming, I use a filter, shown in Figure 3, that places each line of documentation in a comment and moves it to the succeeding code chunk. With this filter, `notangle` transforms a literate program into a traditional commented program, without loss of information and with only a modest penalty in readability.

As a demonstration, and to help convert nuweb programs to `noweb`, I wrote a a 55-line Icon program that makes it possible to abbreviate chunk names using a trailing ellipsis, as in `WEB`; it appears in the `noweb` distribution as `icon/disambiguate.nw`.

Kostas Oikonomou of AT&T Bell Labs and Conrado Martinez-Parra of the Univ. Politecnica de Catalunya in Barcelona have written filters that add prettyprinting to `noweb`. Oikonomou's filters prettyprint Icon and Object-Oriented Turing; Martinez-Parra's filter prettyprints a variant of Dijkstra's language of guarded commands. These filters are in the noweb distribution in the `contrib` directory.

It's also possible to do useful or amusing things by writing new back ends. Figure 4 shows an `awk` script that gives a count of the number of lines of code and of documentation in a group of `noweb` files.

References

Aho, A. V., B. W. Kernighan, and P. J. Weinberger (1988). *The AWK Programming Language*. Reading, MA: Addison-Wesley.

Fraser, C. W. and D. R. Hanson (1995). *A Retargetable C Compiler: Design and Implementation.* Redwood City, CA: Benjamin/Cummings.

Kernighan, B. W. and R. Pike (1984). *The UNIX Programming Environment.* Englewood Cliffs, NJ: Prentice-Hall.

Ramsey, N. (1994, September). Literate programming simplified. *IEEE Software 11*(5), 97–105.

```
BEGIN { bogus = "this is total bogosity"
        codecount[bogus] = -1; docscount[bogus] = -1
      }
/^@file / { thisfile = $2 ; files[thisfile] = 0 }
/^@begin code/ { code = 1 }
/^@begin docs/ { code = 0 }
/^@nl/ {
  if (code == 0)
    docscount[thisfile]++
  else
    codecount[thisfile]++
}
END {
  printf " Code   Docs   Both  File\n"
  for (file in files) {
    printf "%5d  %5d  %5d  %s\n",
        codecount[file], docscount[file],
        codecount[file]+docscount[file], file
    totalcode += codecount[file]
    totaldocs += docscount[file]
  }
  printf "%5d  %5d  %5d  %s\n",
      totalcode, totaldocs, totalcode+totaldocs, "Total"
}
```

Figure 4: Back end for counting lines of code and documentation

The `BEGIN` code forces `codecount` and `docscount` to be associative arrays; without it the increment operator would fail.