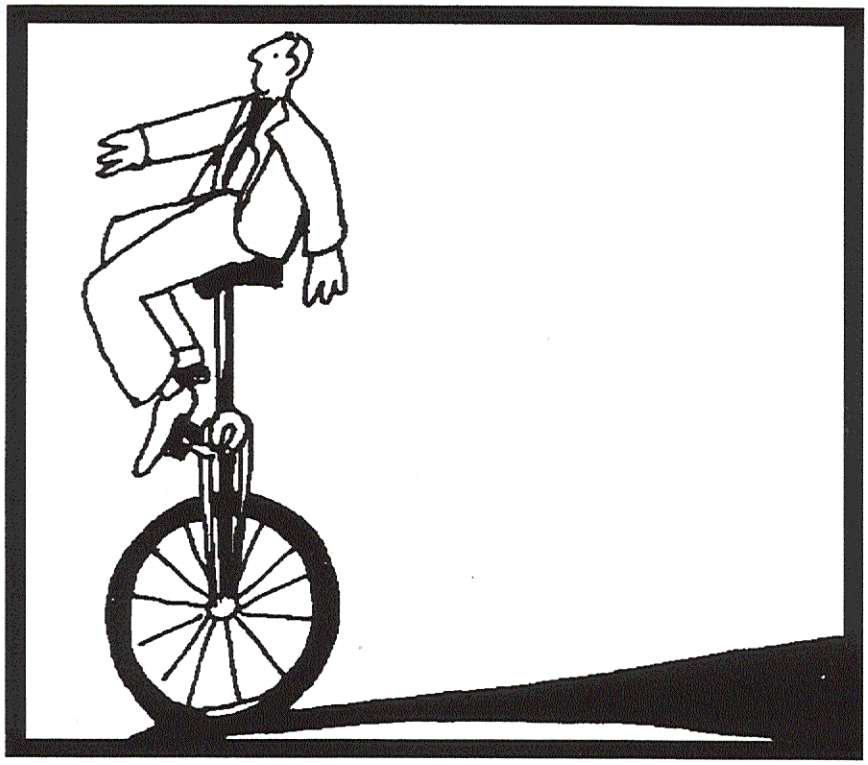
FEATURE

reprinted from IEEE Software

LITERATE PROGRAMMING SIMPLIFIED

Literate programming tools let you arrange the parts of a program in any order and extract documentation and code from the same source file. The author argues that language-dependence and feature complexity have hampered acceptance of these tools, then offers a simpler alternative.



NORMAN RAMSEY
Bellcore

In 1983, Donald Knuth introduced literate programming in the form of Web, his tool for writing literate Pascal programs.[1] Web lets authors interleave source code and descriptive text in a single document. It also frees authors to arrange the parts of a program in an order that helps explain how the program functions, not necessarily the order required by the compiler.

In the mid-'80s, word spread about this new programming method as several literate programs were published. In 1987, *Communications of the ACM* created a special forum to discuss literate programming.[2] Web was adapted to programming languages other than Pascal, including C, Modula-2, Fortran, Ada, and others.[3-6] With experience, however, many Web users became dissatisfied.[7] Continued interest in literate programming led to a frenzy of tool building. In the resulting confusion, the literate-programming forum was dropped, on the grounds that literate programming had become the province of those who could build their own tools.[8]

The proliferation of literate-programming tools made it hard for literate programming to enter the mainstream, but it led to a better understanding of what such tools should do. Today the field is more mature, and there is an emerging demand for tools that are simple, easy to learn, and not tied to a particular programming language.

My own literate-programming tool, noweb, fills this niche. Freely available

0740-7459/94/$04.00 © 1994 IEEE

AN EXAMPLE OF NOWEB: COUNTING WORDS

This example, based on a program by Klaus Guntermann and Joachim Schrod and a program by Silvio Levy and D. E. Knuth, presents the "word count" program from Unix, rewritten in noweb to demonstrate literate programming using noweb. The level of detail in this document is intentionally high, for didactic purposes; many of the things spelled out here don't need to be explained in other programs. The purpose of `wc` is to count lines, characters, and/or words in a list of files. The number of lines in a file is the number of newline characters it contains. The number of characters is the file length in bytes. A word is a maximal sequence of consecutive characters other than newline, space, or tab, containing at least one visible ASCII code. (We assume that the standard ASCII code is in use.)

Most literate C programs share a common structure. It's probably a good idea to state the overall structure explicitly at the outset, even though the various parts could all be introduced in chunks named <*> if we wanted to add them piecemeal.

Here, then, is an overview of the file `wc.c` that is defined by the noweb program `wc.nw`:

*<*98a>≡* *98a*

<Header files to include 98b>
<Definitions 98c>
<Global variables 98d>
<Functions 102d>
<The main program 99a>

Root chunk (not used in this document).

We must include the standard I/O definitions because we want to send formatted output to `stdout` and `stderr`.

<Header files to include 98b>≡ *98b*

```
#include <stdio.h>
```

This code is used in chunk *98a*.

The `status` variable will tell the operating system if the run was successful or not, and `prog_name` is used in case there's an error message to be printed.

<Definitions 98c>≡ *98c*

```
#define OK                    0
  /* status code for successful run */
#define usage_error           1
  /* status code for improper syntax */
#define cannot_open_file      2
  /* status code for file access error */
```

Defines:
`cannot_open_file`, used in chunk *100c*.
`OK`, used in chunk *98d*.
`usage_error`, used in chunk *102d*.
Uses `status` *98d*.
This definition is continued in chunks *100b*, *100e*, and *102c*.
This code is used in chunk *98a*.

<Global variables 98d>≡ *98d*

```
int status = OK;
  /* exit status of command, intially OK */
char *prog_name;
  /* who we are */
```

Defines:
`prog_name`, used in chunks *99a*, *100c*, and *102d*.
`status`, used in chunks *98c*, *99a*, *100c*, and *102d*.
Uses `OK` *98c*.
This definition is continued in chunk *101b*.

on the Internet since 1989, noweb strips literate programming to its essentials. Programs are composed of named chunks of code, written in any order, with documentation interleaved.

To facilitate comparison of Web and noweb, a sample noweb program appears in the shaded box that runs throughout this article. I took the text, code, and presentation for this sample from Knuth's *Literate Programming*.

Noweb was developed on Unix and can be ported to non-Unix platforms provided they can simulate pipelines and support both ANSI C and either awk or Icon. For example, Kean College's Lee Wittenberg ported noweb to MS-DOS. Noweb is unique among literate-programming tools in its pipelined, extensible implementation, which makes it easy for experimenters to create new features without writing their own tools.

WEB'S COMPLEXITIES

Web's complexities make it difficult to explore the *idea* of literate programming because too much effort is required to master the *tool*. To compound the difficulty, different programming languages are served by different versions of Web, each with its own idiosyncrasies.

The classic Web expands three kinds of macros, prettyprints code for typeset output, evaluates some constant expressions, hacks string support into Pascal, and implements a simple form of version control. The manual documents 27 control sequences.[9] Versions for languages other than Pascal offer slightly different functions and different sets of control sequences.

Web uses its Tangle tool to produce source code and its Weave tool to produce documentation. Web's original Tangle removed white space and folded lines to fill each line with

```
Now we come to the general layout of the [[main]]
function.
<<The main program>>=
main(argc, argv)
   int argc;
      /* # arguments on Unix command line */
   char **argv;
      /* the arguments, an array of strings */
{
   <<Variables local to [[main]]>>
   prog_name = argv[0];
   <<Set up option selection>>
   <<Process all the files>>
   <<Print the grand totals if multiple files>>
   exit(status);
}
@ %def main argc argv

If the first argument begins with a `{\tt-}', the
user is choosing the desired counts and specifying
the order in which they should be displayed.
```

***Figure 1.** A noweb source fragment from the example program.*

tokens, making its output unreadable.[1] Later adaptations preserved line breaks but removed other white space. Web's Weave divides a program into numbered "sections," and its index and cross-reference information refer to section numbers, not page numbers. Web works poorly with LaTex: LaTex constructs cannot be used in Web source, and getting Weave output to work in LaTex documents requires tedious adjustments by hand. Weave's source (written in Web) is several thousand lines long, and the formatting code is not isolated.

NOWEB'S FEATURES

Noweb's simplicity derives from a simple model of files, which are marked up using a simple syntax. Figure 1 shows a fragment of the noweb source used to generate the boxed sample program. It shows examples of chunk definitions and uses, quoted code, and lists of defined identifiers — *all* of noweb's syntax except escaped angle brackets.

File structure. A noweb file is a sequence of *chunks*. A chunk may contain code, in which case it is named, or documentation, in which case it is unnamed. Chunks may appear in any order. Each code chunk begins with *<<chunk name>>=* on a line by itself. The double-left angle bracket must be in the first column. Each documentation chunk begins with a line that starts with an @ symbol followed by a space or newline. Chunks are terminated implicitly by the beginning of another chunk or by the end of the file. If the first line in the file does not mark the beginning of a chunk, noweb assumes it is the first line of a documentation chunk.

As Figure 2 shows, noweb uses its notangle and noweave tools to extract code and documentation, respectively. When notangle is given a noweb file, it writes the program on standard output. When noweave is given a noweb file, it reads it and produces, on standard output, Tex source for typeset documentation.

Code chunks. Code chunks contain program source code and references to other code chunks. Several code chunks may have the same name; notangle concatenates their definitions to produce a single chunk

Code-chunk definitions are like macro definitions: Notangle extracts a program by expanding one chunk (by default the chunk named <<*>>). The definition of that chunk contains references to other chunks, which are themselves expanded, and so on. Figure 3 shows part of the boxed sample program as extracted by notangle. Notangle's output is readable; it preserves white space and maintains the indentation of expanded chunks with respect to the chunks in which they appear. This behavior allows noweb to be used with languages like Miranda and Haskell, in which indentation is significant.

When double-left and -right angle brackets are not paired, they are treated as literals. Users can force any such brackets, even paired brackets, to be treated as literal by using a preced-

This code is used in chunk *98a*.

Now we come to the general layout of the `main` function.

<The main program 99a>≡ *99a*

```
main(argc, argv)
  int argc;
    /* # arguments on Unix command line */
  char **argv;
    /* the arguments, an array of strings */
{
  <Variables local to main 99b>
  prog_name = argv[0];
  <Set up option selection 99c>
  <Process all the files 99d>
  <Print the grand totals if multiple files 102b>
  exit (status);
}
```

Defines:
`argc`, used in chunks *99c* and *99d*.
`argv`, used in chunks *99c*, *100c*, and *101e*.
`main`, never used.
Uses `prog_name` *98d* and `status` *98d*.
This code is used in chunk *98a*.

If the first argument begins with a "`-`", the user is choosing the desired counts and specifying the order in which they should be displayed. Each selection is given by the initial character (lines, words, or characters). For example, "`-cl`" would cause just the number of characters and the number of lines to be printed, in that order. We do not process this string now; we simply remember where it is. It will be used to control the formatting at output time.

<Variables local to main 99b>≡ *99b*

```
int file_count;
  /* how many files there are */
char *which;
  /* which counts to print */
```

Defines:
`file_count`, used in chunks *99c*, *100c*, *101e*, and *102b*.
`which`, used in chunks *99c*, *101e*, *102b*, and *102d*.
This definition is continued in chunks *100a* and *100f*.
This code is used in chunk *99a*.

<Set up option selection 99c>≡ *99c*

```
which = "lwc";
  /* if no option is given print 3 values */
if (argc > 1 && *argv[1] == '-') {
  which = argv[1] + 1;
  argc--;
  argv++;
}
file_count = argc - 1;
```

Uses `argc` *99a*, `argv` *99a*, `file_count` *99b*, and `which` *99b*.
This code is used in chunk *99a*.

Now we scan the remaining arguments and try to open a file, if possible. The file is processed and its statistics are given. We use a `do ... while` loop because we should read from the standard input if no file name is given.

<Process all the files 99d>≡ *99d*

```
argc--;
do {
  <If a file is given, try to open *(++argv);
      continue if unsuccessful 100c>
  <Initialize pointers and counters 101a>
  <Scan file 101c>
  <Write statistics for file 101e>
  <Close file 100d>
  <Update grand totals 102a>
```

```
    /* even if there is only one file*/
} while (--argc > 0);
```

Uses **argc** *99a*.
This code is used in chunk *99a*.

Here's the code to open the file. A special trick allows us to handle input from **stdin** when no name is given. Recall that the file descriptor to **stdin** is 0; that's what we use as the default initial value.

<Variables local to **main** *99b>+≡* *100a*

```
int fd = 0;
  /* file descriptor, initialized to stdin */
```

Defines:
fd, used in chunks *100c*, *100d*, and *101d*.

<Definitions 98c>+≡ *100b*

```
#define READ_ONLY 0
  /* read access code for system open */
```

Defines:
READ_ONLY, used in chunk *100c*.

<If a file is given, try to open ***(++argv);** **continue** *if unsuccessful 100c>≡* *100c*

```
if (file_count > 0
&& (fd = open (*(++argv),READ_ONLY)) < 0) {
  fprintf(stderr,
    "%s: cannot open file %s\n",
    prog_name, *argv);
  status |= cannot_open_file;
  file_count--;
  continue;
}
```

Uses **argv** *99a*, **cannot_open_file** *98c*, **fd** *100a*, **file_count** *99b*, **prog_name** *98d*, **READ_ONLY** *100b*, and **status** *98d*.
This code is used in chunk *99d*.

<Close file 100d>≡ *100d*

```
close (fd);
```

Uses **fd** *100a*.
This code is used in chunk *99d*.

We will do some homemade buffering in order to speed things up: Characters will be read into the **buffer** array before we process them. To do this we set up appropriate pointers and counters.

<Definitions 98c>+≡ *100e*

```
#define buf_size BUFSIZ
  /* stdio.h BUFSIZ chosen for efficiency */
```

Defines:
buf_size, used in chunks *100f* and *101d*.

<Variables local to main 99b>+≡ *100f*

```
char buffer[buf_size];
  /* we read the input into this array */
register char *ptr;
  /* first unprocessed character in buffer */
register char *buf_end;
  /* the first unused position in buffer */
register int c;
  /* current char, or # of chars just read */
int in_word
  /* are we within a word? */
long word_count, line_count, char_count;
  /* # of words, lines, and chars so far */
```

Defines:
buf_end, used in chunks *101a* and *101d*.
buffer, used in chunks *101a* and *101d*.
char_count, used in chunks *101a*, *101d*, *101e*, *102a*, and *102d*.
in_word, used in chunks 101a and 101c.

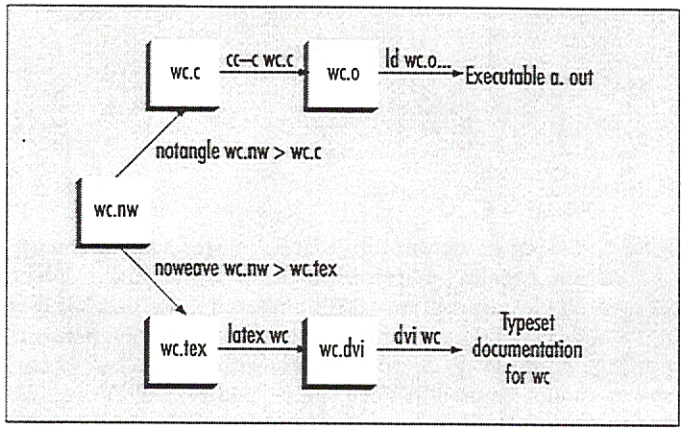


Figure 2. Using noweb to build code and documentation.

ing @ sign.

Any line beginning with @ and a space terminates a code chunk. If such a line has the form `@□%def` *identifiers* it also means that the preceding chunk defines the identifiers listed in *identifiers*. This notation provides a way of marking definitions manually when no automatic marking is available.

Documentation chunks. Documentation chunks contain text that is ignored by notangle and copied verbatim to standard output by noweave (except for quoted code). Code may be quoted within documentation chunks by placing double square brackets around it. These brackets are ignored by notangle but are used by noweave to give the quoted code special typographic treatment. For example, in the sample program, quoted code is set in the Courier font.

Noweave can work with LaTex, or it can use a plain Tex macro package, supplied with noweb, that defines commands like `\chapter` and `\section`. Noweave can also work with HTML, the hypertext markup language for Mosaic and the World-Wide Web. The example simulates the results after processing by noweave and LaTex.

Noweave adds no newline characters to its output, making it easy to find the sources of Tex or LaTex errors. For example, an error on line 634 of a generated Tex file is caused by a problem on line 634 of the corresponding noweb file.

Index and cross-reference features. Cross-referencing of chunks and identifiers makes large programs easier to understand. The sample program accompanying this article shows full cross-reference information.

Unlike Web, noweb does not introduce numbered "sections" for cross-referencing. Noweb uses page numbers. If two or more chunks appear on a page, say page 24, they are distinguished by appending a letter to the page number: 24a or 24b, for example. Readers of large literate programs will appreciate the use of a single numbering system.

Like Web, noweb writes chunk-cross-reference information in a footnote font below each code chunk. Noweb also includes cross-reference information for identifiers, for example, `Defines file_count, used in chunks 99c, 100c, 101e, and 102b`. Noweb generates this by using the `@□%def` markings in its source code, or by recognizing definitions automatically. Although noweb can automatically recognize definitions in C programs, I used `@□%def` to mark the definitions in the sample program. This choice not only illustrates the use of `@□%def` but

it also ensures results compatible with the CWeb version of this program. Automatically generated indices would differ because CWeb and noweb use different recognition heuristics. Because noweb uses a language-independent heuristic to find identifier uses, it can be fooled into finding false "uses" in comments or string literals, like the use of `status` in chunk *98c*.

Compiler and debugger support. On a large project, it is essential that compilers and other tools refer to locations in the noweb source, even though they work with notangle's output.[8] Giving notangle the `-L` option makes it emit pragmas that inform compilers of the placement of lines in the noweb source. It also preserves the columns in which tokens appear, so that line-and-column error messages are accurate. If you do not give notangle the `-L` option, it respects the indentation of its input, making its output easy to read.

Formatting features. Noweave depends on text formatters in two ways: in the source of noweave itself and in the supporting macros. Noweave's dependence on its formatter is small and isolated, instead of being distributed throughout a large implementation. Noweb uses 250 lines of source for Tex and LaTex combined, and another 250 for HTML. It uses about 200 lines of supporting macros for plain Tex and another 300 lines to support LaTex, primarily because the page-based cross-reference mechanism is complex. LaTex support without cross-ref-

```
main(argc, argv)
   int argc;
      /* # arguments on Unix command line */
   char **argv;
      /* the arguments, an array of strings */
{
   int file_count;
      /* how many files there are */
   char *which;
      /* which counts to print */
   int fd = 0;
      /* file descriptor, initialized to stdin */
   char buffer[buf_size];
      /* we read the input into this array */
   register char *ptr;
      /* first unprocessed character in buffer */
   register char *buf_end;
      /* the first unused position in buffer */
   register int c;
      /* current char, or # of characters just read */
   int in_word;
      /* are we within a word? */
   long word_count, line_count, char_count;
      /* # of words, lines, and characters so far */
   prog_name = argv[0];
   which = "lwc";
```

Figure 3. Part of the example program after extraction by notangle.

`line_count`, used in chunks *101a*, *101c*, *101e*, *102a*, and *102d*.
`ptr`, used in chunks *101a*, *101c*, and *101d*.
`word_count`, used in chunks *101a*, *101c*, *101e*, *102a*, and *102d*.
Uses `buf_size` *100e*.

<Initialize pointers and counters 101a>≡ *101a*

```
ptr = buf_end = buffer;
line_count = word_count = char_count = 0;
in_word = 0;
```

Uses `buf_end` *100f*, `buffer` *100f*, `char_count` *100f*, `in_word` *100f*, `line_count` *100f*, `ptr` *100f*, and `word_count` *100f*.
This code is used in chunk *99d*.

The grand totals must be initialized to zero at the beginning of the program. If we made these variables local to `main`, we would have to do this initialization explicitly; however, C's globals are automatically zeroed. (Or rather, "statically zeroed.") (Get it?)

<Global variables 98d>+≡ *101b*

```
long tot_word_count, tot_line_count,
    tot_char_count;
  /* total number of words, lines, chars */
```

The present chunk, which does the counting that is `wc`'s *raison d'être*, was actually one of the simplest to write. We look at each character and change state if it begins or ends a word.

<Scan file 101c>≡ *101c*

```
while (1) {
  <Fill buffer if it is empty; break at end of file 101d>
  c = *ptr++;
  if (c > ' ' && c < 0177) {
    /* visible ASCII codes */
    if (!in_word) {
      word_count++;
      in_word = 1;
    }
    continue;
  }
  if (c == '\n') line_count++;
  else if (c != ' ' && c != '\t') continue;
  in_word = 0;
    /* c is newline, space, or tab */
}
```

Uses `in_word` *100f*, `line_count` *100f*, `ptr` *100f*, `word_count` *100f*.
This code is used in chunk *99d*.

Buffered I/O allows us to count the number of characters almost for free.

<Fill buffer if it is empty; break at end of file 101d>≡ *101d*

```
if (ptr >= buf_end) {
  ptr = buffer;
  c = read(fd, ptr, buf_size);
  if (c <= 0) break;
  char_count += c;
  buf_end = buffer + c;
}
```

Uses `buf_end` *100f*, `buffer` *100f*, `buf_size` *100e*, `char_count` *100f*, `fd` *100a*, and `ptr` *100f*.
This code is used in chunk *101c*.

It's convenient to output the statistics by defining a new function `wc_print`; then the same function can be used for the totals. Additionally we must decide here if we know the name of the file we have processed or if it was just `stdin`.

<Write statistics for file 101e>≡ *101e*

```
wc_print(which, char_count, word_count,
        line_count);
if (file_count)
```

```
    printf(" %s\n", *argv); /* not stdin */
  else
    printf ("\n");          /* stdin */
```

Uses argv *99a*, char_count *100f*, file_count *99b*, line_count *100f*, wc_print *102d*, which *99b*, word_count *100f*.

This code is used in chunk *99d*.

<Update grand totals 102a>≡ *102a*

```
tot_line_count += line_count;
tot_word_count += word_count;
tot_char_count += char_count;
```

Uses char_count *100f*, line_count *100f*, word_count *100f*.

This code is used in chunk *99d*.

We might as well improve a bit on Unix's wc by displaying the number of files too.

<Print the grand totals if multiple files 102b>≡ *102b*

```
if (file_count > 1) {
  wc_print(which, tot_char_count,
           tot_word_count, tot_line_count);
  printf("total in %d files\n",file_count);
}
```

Uses file_count *99b*, wc_print *102d*, which *99b*.

This code is used in chunk *99a*.

The function below prints the values according to the specified options. The calling routine should supply a newline. If an invalid option character is found we inform the user about proper use of the command. Counts are printed in eight-digit fields so they will line up in columns.

<Definitions 98c>+≡ *102c*

```
#define print_count(n) printf("%8ld", n)
```

Defines:
print_count, used in chunk *102d*.

<Functions 102d>≡ *102d*

```
wc_print(which, char_count, word_count,
         line_count)
  char *which; /* which counts to print/
  long char_count, word_count, line_count;
    /* given totals */
{
  while (*which)
    switch (*which++) {
      case 'l': print_count(line_count);
        break;
      case 'w': print_count(word_count);
        break;
      case 'c': print_count(char_count);
        break;
      default:
        if ((status & usage_error) == 0) {
          fprintf(stderr,
            "\nUsage: %s [-lwc] [filename ...]\n",
              prog_name);
          status |= usage_error;
        }
    }
}
```

Defines:
wc_print, used in chunks *101e* and *102b*.
Uses char_count *100f*, line_count *100f*, print_count *102c*, prog_name *98d*, status *98d*, usage_error *98c*, which *99b*, and word_count *100f*.
This code is used in chunk *98a*.

A test of this program against the system wc command on a SparcStation showed the "official" wc was slightly slower. Although that wc gave an appropriate error message for the options -abc, it made no complaints about the options -labc! Dare we suggest the system routine might have been better had its programmer used a more literate approach?

erencing requires only 34 lines of source and no supporting macros. HTML requires no supporting macros.

Uncoupling files and programs. The mapping between noweb files and programs is many-to-many; the mapping between files and documents is many-to-one. You combine source files by listing their names on notangle's or noweave's command line. Notangle can extract more than one program from a single source file by using the -R command-line option to identify the root chunks of the different programs.

The simplest example of one-to-many program mapping is that of putting a C header and program in a single noweb file. The header comes from the root chunk *<header>*, and the program from the default root chunk, <*>. The following Unix commands extract files wc.h and wc.c from noweb file wc.nw.

```
notangle -L wc.nw > wc.c
notangle -Rheader wc.nw | cpif -ne wc.h
```

The > in the first command directs notangle's output to the file wc.c. The | in the second command directs notangle's output to the cpif program, which is distributed with noweb. `cpif -ne wc.h` compares its input to the contents of file wc.h; if they differ, the input replaces wc.h. This trick avoids touching the file wc.h when its contents have not changed, which avoids triggering unnecessary recompilations.

Because it is language-independent, noweb can combine different programming languages in a single literate program. This ability makes it possible to explain *all* of a project's source in a single document, including not just ordinary code but also things like make files, test scripts, and test inputs. Using literate programming to describe tests as well as source code provides a lasting, written explanation of the thinking needed to create the tests, and it does so with little overhead. If not documented at the time, the rationale behind complex tests can easily be lost.

IMPLEMENTING NOWEB

Until now we have discussed noweb from a user's point of view, showing that it is simple and easy to use. Noweb's implementation is also worth discussing, because noweb's extensible implementation makes it unique among literate-programming tools. 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.

Users change or extend noweb not by recompiling but by inserting or removing pipeline stages; for example, noweave switches from LaTex to HTML by changing just the last pipeline stage. Noweb's extensibility enables its users to create new literate-programming features without having to write their own tools.

Noweb's syntax is easy to read, write, and edit, but it is not easily manipulated by programs. Markup, which is the first stage in every pipeline, converts noweb source to a representa-

tion easily manipulated by common Unix tools like sed and awk, greatly 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.

In the pipeline representation, every line begins with @ and a keyword. The most important possibilities appear in Table 1. Markup brackets chunks by @begin ... @end, and it uses the noweb source to identify text and newlines, definitions and uses of chunks, and quoted code, which can all appear inside chunks. It also preserves information about file names and defined identifiers. Other index and cross-reference information is inserted automatically by later pipeline stages. The details of noweb's pipeline representation are described in the *Noweb Hacker's Guide*, which is distributed with noweb.

EXTENDING NOWEB

Noweb lets users insert stages into the notangle and noweave pipelines, so that they can change a tool's existing behavior or add new features without recompiling. Even language-dependent features like formatted output and automatic index generation have been added to noweb without recompiling.

Stages inserted in the middle of a pipeline both read and write noweb's pipeline representation; they are called *filters*, by analogy with Unix filters, which are used in the Unix implementation.

Filters can be used to change the way noweb works; for example, a one-line sed script makes noweb treat two chunk names as identical if they differ only in their representation of white space, as in Web. A 55-line Icon program makes it possible to abbreviate chunk names using a trailing ellipsis. To share programs with colleagues who don't enjoy literate programming, I use a filter 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.

Filters can be used to add significant features. Noweave's cross-reference and indexing features use two filters, one that finds uses of defined identifiers and one that inserts cross-reference information. In most cases, programmers must mark identifier definitions by hand, using `@␣%def`, but in some cases a third, language-dependent filter can be used to mark identifier definitions, making index generation completely automatic.

Kostas Oikonomou of AT&T Bell Labs, Kaelin Colclasure of Bridge Information Systems, and Conrado Martinez-Parra of the Universidad Politecnica de Catalunya in Barcelona have written noweb filters that add prettyprinting for Icon, C++, and Dijkstra's language of guarded commands, respectively.

Noweb turns a World-Wide-Web browser like Mosaic into a hypertext browser for literate programs. For example, you can click on an identifier or chunk name to jump to the definition of that identifier or chunk. You can find a hypertext version of the boxed sample program at ftp://bellcore.com/pub/norman/noweb/wc.html.

EVALUATING NOWEB

Reviewers have had many expectations of literate-programming tools.[8, 10] We expect to be able to write code chunks in any order. We expect to develop code and documentation in one place. Finally, we expect automatically generated cross-reference and index information. Like the original Web, noweb provides all these features, but in simpler form.

Web does provide features that noweb lacks, but existing Unix tools can substitute for most of these. Although noweb contains no internal support for macros, Unix supplies two macro processors that can work with noweb: the C preprocessor and the m4 macro processor. The xstr program extracts string literals, and the patch program provides a form of version control similar to Web's change files.

Indexing and cross-referencing make noweb less simple than it could be. I need complex LaTex code to compute page numbers for use in cross-reference lists and in the index. The ability to use page numbers justifies this com-

TABLE 1
NOWEB PIPELINE REPRESENTATION

Keyword	Definition
@begin *kind*	Start a chunk
@end *kind*	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 document chunk
@endquote	End of quoted code in a document chunk
@file *filename*	Name of the file from which the chunks came
@index defn *ident*	The current chunk contains a definition of *ident*
@index ...	Automatically generated index information
@xref ...	Automatically generated cross-reference information

plexity, especially since it can be hidden from most users. You do need to understand the LaTex code if you want to customize the appearance of your noweb documents while retaining noweb's use of page numbers for cross-reference. Most literate-programming tools forbid customization, but not all users will accept such a restriction. I have compromised between simplicity and customizability by adding LaTex options for a dozen of the most commonly requested customizations. Users can choose from among these options without understanding noweb's LaTex code.

Experimenting with noweb is easy because the tools are simple. If the experiment is unsatisfying, it is easy to abandon, because notangle's output is readable, and documentation can be preserved as embedded comments. Noweb is simpler than Web and easier to use and understand, but it does less. I argue, however, that the benefit of Web's extra features is outweighed by the cost of the extra complexity, making noweb better for writing literate programs. Few of Web's remaining features will be missed; for example, many compilers evaluate constant expressions at compile time. Noweb users are most likely to miss pretty-printing, but it may be more trouble than it is worth.[7]

In my own work, I have used noweb for code written in various languages, including assembly language, awk, Bourne shell, C, Icon, Modula-3, Promela, Standard ML, and Tex. These projects have ranged in size from a few hundred to twenty thousand lines of code. Information about other programs written using noweb is hard to find. Noweb is provided free of charge, generating no sales records. Programs created with noweb may be delivered in the form of ordinary source code, leaving no clue that noweb was used. The only way for me to find out about uses of noweb is to appeal for information on the Internet. In this way I have learned about significant noweb projects in C++, Modula-2, Occam, parallel C, Perl, Prolog, and Scheme.

LANGUAGE-INDEPENDENT TOOLS LIKE NOWEB ARE SIMPLER AND EASIER TO USE THAN TRADITIONAL COMPLEX TOOLS.

David Hanson and Chris Fraser are using noweb to write a book describing the design and implementation of a retargetable C compiler. Tipton Cole & Company use Noweb in their consulting business, which focuses on writing database applications on DOS platforms. They find that noweb helps compensate for some of the deficiencies in DOS database tools, and that literate programming helps when a customer requests a change in a program that hasn't been touched in a year. A customer-support group at Sun Microsystems is using noweb to help teach their customers how to work with aspects of the Solaris operating system like threads and device drivers. The literate-programming paradigm makes it possible to extract working code from the same source used to create technical reports and newsletters.

OTHER TOOLS

A survey of literate-programming tools is beyond the scope of this article, but we can still sketch noweb's place in the context of other tools. Most literate-programming tools are language-dependent and complex. You must change tools when changing programming languages, repeating effort spent mastering a tool.

Newer tools, like noweb, are language-independent. The three most prominent are noweb, nuweb, and Funnelweb.

To users, Noweb and nuweb look very similar. There are minor syntactic differences, and nuweb uses markup within the source file instead of command-line options to show things like the names of output files, but both are simple and easy to master. Funnelweb is a complex tool that includes its own rudimentary typesetting language and command shell.

Many of the similarities between noweb and nuweb arise by design. Nuweb's initial design borrowed from noweb, and later versions of each tool have incorporated ideas from the other.

Noweb and nuweb differ substantively in implementation. Nuweb is not pipelined; it is a single, monolithic C program. This structure makes nuweb easy to port, since only a C compiler is needed, and it makes it faster, since no parts are interpreted and the overhead of creating a pipeline is eliminated, but it also makes nuweb hard to extend. Noweb's pipeline makes it easy to extend, and different stages of the pipeline can be implemented in different programming languages, depending on which language is best for which job. Extensibility is particularly valuable to those interested in pushing the frontiers of literate programming, who would otherwise have to write their own tools from scratch.

I advocate language-independent tools for two reasons. First, after mastering one such tool, you can write almost anything as a literate program, including things like shell and perl scripts, which often benefit disproportionately from a literate treatment. Second, two of these tools — noweb and nuweb — are much simpler, and therefore much easier to master, than any of the language-dependent tools. Those who use one language exclusively may, however, prefer a language-dependent tool, since it provides pretty-printing, which when done well can make the printed literate program easier to read.

Noweb probably culminates one kind of evolution in literate programming: the trend toward greatest simplicity. No significantly simpler tool could do much. Noweb also begins another kind of evolution, toward greater extensibility and flexibility. Further evolution might involve replacing Unix shell scripts and pipelines with an embedded language having special data types to represent pipelines, chunks, and literate programs. This step would make it easier to port noweb to nonUnix platforms, and it could make noweb run much faster. Other developments might include constructing new pipeline stages to support language-dependent operations like macro processing, pretty-printing, and automatic identifier cross-reference.

These changes would extend noweb's capabilities, but noweb is already quite capable of supporting complex programs and documents. It and related tools are less capable of supporting a modern word-processing style. The word processors noweb currently supports, Tex, LaTex, and HTML, all use the old batch model of word processing. Today, many authors prefer WYSIWYG word processors like Framemaker, WordPerfect, or Microsoft Word. Kean College's Wittenberg has developed a noweb-like system called WinWordWeb based on Word. Because of Word's limitations, including its secret proprietary data format, he could not reuse any of noweb's implementation, but the design is the same.

The challenge for literate programming today is getting it into use. Noweb helps by eliminating clutter and complexity. Supporting modern word processors would eliminate another barrier, making it possible to write literate programs without first learning a new word-processing language like LaTex or HTML.

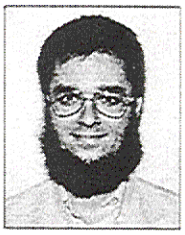
More must be learned about suitable ways of structuring literate programs, about whether hypertext is a useful alternative, and about what other kinds of documents literate programs should resemble. What place does literate programming have for the majority of programmers, who are not writing for publication? In the near term, I suspect the best use for literate programming will be to support rapid prototyping, providing a simple and reliable way of documenting the design decisions made in, and the lessons learned from, the prototype. In the long term, I hope that simple, extensible tools like noweb will lead everyone to appreciate the benefits of literate programming. ◆

ACKNOWLEDGEMENTS

Mark Weiser's invaluable encouragement provided the impetus for me to write this paper, which I did while visiting the Computer Science Laboratory of the Xerox Palo Alto Research Center. David Hanson suggested and provided the cpif program. Preston Briggs developed many of the ideas used in noweb's indexing, and he contributed code used in one of the pipeline stages. Bill Trost wrote the first HTMLpipeline stage. Dave Love provided much-needed LaTex expertise. Comments from Hanson and from the anonymous referees stimulated me to improve the paper. The development of noweb was supported by a Fannie and John Hertz Foundation Fellowship.

REFERENCES

1. D.E. Knuth, *Literate Programming*, Stanford University, Stanford, Calif., 1992.
2. P.J. Denning, "Announcing Literate Programming," *Comm. ACM*, July 1987, p. 593.
3. K. Guntermann and J. Schrod, "Web Adapted to C," *TUGBoat*, Oct. 1986, pp. 134-137.
4. S. Levy, "Web Adapted to C, Another Approach," *TUGBoat*, April 1987, pp. 12-13.
5. N. Ramsey, "Literate Programming: Weaving a Language-Independent Web," *Comm. ACM*, Sept. 1989, pp. 1051-1055.
6. H. Thimbleby, "Experiences of 'Literate Programming' Using CWeb (a Variant of Knuth's Web)," *Computer Journal*, 1986, pp. 201-211.
7. N. Ramsey and C. Marceau, "Literate Programming on a Team Project," *Software — Practice & Experience*, July 1991, pp. 677-683.
8. C. J. Van Wyk, "Literate Programming: An Assessment," *Comm. ACM*, Mar. 1990, pp. 361-365.
9. D.E. Knuth, "The Web System of Structured Documentation," Tech. Report 980, Computer Science Dept., Stanford Univ., Stanford, Calif., 1983.



Norman Ramsey is a research scientist at Bellcore. His research interests are the construction of software that is easy to understand and to retarget to different machines. His recent work includes a retargetable debugger and a toolkit that helps build debuggers and other programs that manipulate machine code.

Ramsey received a PhD in computer science from Princeton University. He is a member of ACM.

Address questions about this article to Ramsey at Bellcore, 445 South Street, Morristown, NJ 07960; norman@bellcore.com. Noweb can be obtained by anonymous ftp from CTAN, the Comprehensive Tex Archive Network, in directory web/noweb. CTAN replicas appear on hosts ftp.shsu.edu, ftp.tex.ac.uk, and ftp.uni-stuttgart.de. Noweb's World-Wide-Web page is located at ftp://bellcore.com/pub/norman/noweb.