LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - The University of Edinburgh

Literate Programming: A Review

by

Angus Duggan

LFCS Report Series ECS-LFCS-93-263

LFCS April, 1993

Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

Copyright © 1993, LFCS

Copyright © 1993, Laboratory for Foundations of Computer Science
University of Edinburgh. All rights reserved.

Reproduction of all or part of this work is
permitted for educational or research use on
condition that this copyright notice is included
in any copy.

Literate Programming: A Review

Angus Duggan

Department of Computer Science
The University of Edinburgh
The King's Buildings
Mayfield Road
Edinburgh EH9 3JZ

email: ajcd@dcs.ed.ac.uk

Tuesday 27th April 1993

Abstract

The *literate programming* concept was formulated by D. E. Knuth in 1984 [45]. Since then, there has been relatively little interest in the system which Knuth described (with reference to the TEX project) as "the most important thing about all this research" [43]. This article is a review of the literate programming concept, and a survey of research in the area.

Some of the pieces of related work which have used experimental results to guide the design of literate programming tools will also be reviewed.

1 Introduction

This report is based on a colloquium given at the National Physical Laboratories [32] using material from a literature survey done for the author's Ph.D.

Literate Programming is a term coined by D. E. Knuth in 1984, to describe the programming methodology used to create the TEX and METAFONT programs. The term *literate programming* reflects the nature of the methodology — that by considering programs as works of literature the structure of the program and the accompanying documentation can be improved. Knuth also admits to some malice in the name of the methodology; after being forced to adopt structured programming because he didn't want to be accused of *unstructured* programming, Knuth has got his own back by making other people consider whether they are writing *illiterate* programs.

The key idea behind literate programming is that instead of just instructing the *computer* what to do, the programmer concentrates on explaining to *human beings* what the computer is to do.

Literate programming allows the programmer to combine the best features of top-down and bottom-up program design methods. Top-down design imparts a strong sense of direction to the programming, but keeps a lot of details in suspense until the end. Bottom-up design continually improves the toolkit with which the program is being built, but postpones the overall program organisation to the last minute. Literate programming creates the program

in a "stream of consciousness" order, which allows sections of the program to be developed top-down or bottom-up, whichever is most appropriate to the exposition of the program.

Knuth developed a tool, WEB, which integrates the documentation and code of a program into a single file. WEB provides macro-processing features which facilitate the "stream of consciousness" order of presentation. It was intended to make it easier for experienced programmers to develop and document programs. He quotes Anthony Hoare:

> "Documentation must be regarded as an integral part of the process of design and coding. A good programming language will encourage and assist the programmer to write clear, self-documenting code, and even perhaps to develop and display a pleasant style of writing."

The best known works which use literate programming are Knuth's text formatter, TEX [46], and algorithmic font generator METAFONT [47]. These programs, along with their supporting utility programs, constitute a large body of software which was the reason for WEB's creation. The success of TEX has been due in some extent to the ease with which it has been ported to new machines, which in turn is due to it being written in WEB.

2 WEB

Knuth's original WEB system (as used for TEX and METAFONT) is a macro pre-processing system tailored for the Pascal programming language and the TEX typesetting system [44]. A single source file contains "modules"[1] which consist of a *documentation* part, a *definition* part, and a *code* part.

The code part of the module contains a fragment of program, which is associated with a name. References to this name can be made in other code parts, indicating that the fragment associated with the name is to be inserted at that point in the program. The features of WEB macros which make them useful for explaining programs are:

- More than one association can be made to each name—all of the fragments associated with the same name are collected together before expansion. This allows pieces of code which have syntactic constraints on their positioning in the program to be separated out and explained in juxtaposition to related pieces of code.

- The names of code parts in the WEB system can be whole sentences, containing nearly any sequence of characters desired. References to names can be abbreviated, as long as the abbreviation is unambiguous, for the convenience of the programmer.

Un-named code parts are also permitted: these are collected together and used as the root module of the program, which will be expanded to create the whole program source code.

The definition part of a module is used for macro definitions which are expanded in the code parts. These macros are a single line of text, optionally with one parameter which is substituted in the replacement text. The definition parts of WEB were included mainly to make up for deficiencies in the Pascal programming language.

[1] The choice of the term "modules" is an unfortunate one in retrospect, because WEB modules are not at all similar to the more common meaning of modules in Computer Science—that of separately compiled units of a program with a well-defined public interface and private implementation.

The documentation part of a module is intended as a commentary explaining what the code and definition parts are used for. There is a very rudimentary document structuring facility provided by "major modules", which are made more visually prominent.

There are two programs in the WEB system, both of which operate on the single source file: TANGLE takes the source file and produces a compilable Pascal program from it, and WEAVE takes the source file and produces a typesettable TEX document from it. The program listing produced by TANGLE is deliberately obfuscated by removing comments and non-significant whitespace, with the intention of forcing the programmer to use the typeset listing. The WEAVE program which produces typeset listings of the program incorporates a sophisticated, but unfortunately not very customisable, pretty-printer for the Pascal code sections. Knuth's ideas of good program layout do not suit everybody, and may be one of the reasons why programmers are put off using WEB. Automatic cross-referencing and indexing of program modules and identifiers in the program is also performed by WEAVE.

WEB does not make any attempt to hide the details of typesetting commands from the user, and indeed adds another (albeit simple) macro language to the systems that the user must master to produce elegant programs. A simple "changefile" facility is provided for merging system-dependent changes into the program. Changefiles define a list of paragraphs which should be taken out of the program, with altered versions which should replace them.[2]

3 Extensions to WEB

Extensions to the WEB system to make it more useful in "real-world" situations have been made by several people. Many of these extensions have been alterations to the change file mechanism, to cope with multiple changefiles.

Damerell [29] altered WEB's program pre-processor (Tangle) to increase the level of error detection. Some difficult errors to trace in both the WEB macro language and the Pascal program code are detected. In addition, errors were reported to a log file as well as to the terminal.

Appelt and Horn [3] extended WEB to use multiple changefiles for programs with multiple system-dependent changes and revision levels. They point out that this facility is most useful when changes are orthogonal, but they do allow the possibility of prioritising changes which affect the same part of the original source file.

Both Guntermann and Rülling [35], and Sewell [77] wrote separate preprocessors to allow the use of multiple changefiles with WEB. They took different approaches to changefile conflicts: Sewell's approach in his WEBMERGE program was similar to Appelt and Horn's, in which one changefile took precedence over the others, and a warning of a conflict is issued. Guntermann and Rülling's program, TIE, applied each changefile successively, as if the previous changefiles had been merged with the original program. This approach is more suitable when changefiles are being used for development of new versions of a program, where the new code is developed as changefiles without altering the original program, and then is merged in to create a new release version. Both of these pre-processors create a new WEB file generated by merging the original program and changefiles.

Breitenlohner's PATCH processor [14] takes the same approach to merging multiple changefiles as TIE, but also has a file insertion feature; a control sequence is used to indicate that a patch is to be inserted in the generated WEB file. This feature allows common sections of code to be factored out of programs which use them, and maintained as one source file. PATCH has

[2] The idea is similar to the *context* `diff` found on UNIX systems today.

three modes of operation: in *merge* mode it creates a single WEB file and changefile, suitable for feeding to WEAVE and TANGLE. In *insert* mode PATCH produces one WEB file containing all changes and insertions, and in *update* mode all changes are applied to the primary patch, but insertions are not made; this mode is applicable for creating new release versions once all changes have been fully tested. Breitenlohner found the PATCH code useful enough to include in TANGLE and WEAVE, eliminating the need for an extra pre-processing step.

4 WEBs for other languages

There have been several adaptations of WEB to other programming languages, including C, Smalltalk, Modula-2, FORTRAN, Ada, C++, Scheme, and Reduce.

The first adaptation of WEB to another programming language was Harold Thimbleby's **cweb** system, created in 1984 [83]. This system was designed before Knuth's article in The Computer Journal [45] appeared—the author heard Knuth give a talk about literate programming, and implemented his own system from the ideas presented in the talk. Thimbleby used **troff** as the formatter for his system, but also included a cut-down formatter in the system for quick-and-dirty draft printouts.

Thimbleby's **cweb** system does not include the more sophisticated pretty-printing functions of Knuth's WEB, but does allow the programmer to lay the code out in the way he wants. Some manual intervention is necessary to create good-looking listings with Thimbleby's system. A simple alignment facility is provided to insert markers for aligning fields in the code listing, but no parsing of the program code is performed for pretty-printing. The lack of sophisticated parsing of the program code means that an index of identifiers could not be generated automatically.

Cweb detected when changes were made to the actual code in a file (not including the comments), and re-wrote the output file only when a substantive change was made, allowing `make` to work more effectively on these files. Included headers are supported by Thimbleby's **cweb**.

Thimbleby later reported experiences of using **cweb** [82], including subjective evidence that students forced to use **cweb** produced better programs and reports for assignments than those who did not use it. He states that:

> "In the past program listings were submitted for examination as subsidiary material, which was usually extensive, superficially homogeneous and disorganised in comparison with the written project report. The project report would also contain extensive reference to the program and often laborious internal documentation. Those students who have used **cweb** have presented informative programs integral to the project report, and the previous tedious internal documentation is in one place and is easier to mark."

He examined the possibility of including literate programming systems in Integrated Project Support Environments (IPSEs), and mentions some of the possibilities for WEB manipulation tools, but did not pursue this line of work any further.

Three more adaptations of WEB for C were made by Guntermann and Schrod [36, 75], Levy [54], and Dong [95]. Levy's version translates WEB's definition sections into C pre-processor directives, which are then processed as usual by the compiler. He also introduced an "include file" directive, allowing CWEB programs to be split up into multiple files. The tangled output of Levy's version included `#line` directives which are recognised by most C compilers, allowing

compilation errors to be referred directly to the line in the CWEB source file, rather than the intermediate tangled file. Levy's version of WEB for C has been developed by himself and Knuth to handle C++ constructs, and is becoming the de-facto standard WEB for C.

Dong's CDS literate programming system combines the C programming language with a language called SP for typesetting Chinese and English.

Sewell [76] created a literate programming system for Modula-2, called MANGLE. The changes from the original Pascal-based WEB were easy enough that the implementation was done in changefiles. The MANGLE system is used for examples in his book about literate programming [78].

A specialised application of WEB for Modula-2 was demonstrated by Kredel [50]. He created a translator to convert routines for a computer algebra system into Modula-2 routines using WEB, with a view to using Sewell's MWEB if it is available.

A version of WEB for Ada was created by Wu and Baker [94]. Their conclusions after using it were that it was a far superior method of producing documentation, but there were some problems with WEB as implemented. In particular the extra directives required to produce good-looking code were awkward, the lack of hierarchical structure in the document frustrated good explanation of the program, the requirement for code segment names to be used before defined in WEB was not suitable for writing re-usable components and that explanatory code which is not part of the program cannot be formatted the same way as the code. These criticisms are echoed by some other authors, and may be due to the original WEB being over specialised.

Adaptations of WEB for Ada (again) [66], Scheme [68], Fortran [6, 5] and C++ [41] have also been done, with varying degrees of success.

Reenskaug and Skaar's adaptation of literate programming to Smalltalk is interesting, because it integrates the concepts of literate programming to work within the Smalltalk environment and idiom.

The Smalltalk environment allows integration of code and documentation as usual, but also allows easy inclusion of diagrams and figures in the documentation. An interactive "galley editor" gives a graphical overview of the structure of the document, with icons representing text, code, figures and pictures. Insertions, deletions, and re-arrangements can be performed on the document by manipulating the icons corresponding to sections, using the common cut and paste idiom found in many GUIs. The structure of the document can include nested sub-sections, to any level required.

Facilities are provided to browse the Smalltalk program library, and insert the code of objects into the editor. The code fragments in the editor can be compiled interactively, and placed back in the program library if desired. This does restrict the code fragments allowable in sections to compilable Smalltalk classes or methods. One of the "paradigmatic" features of literate programming (see section 5) is removed by this restriction; the ability to refine the program by using arbitrary names to represent code which has not been explained yet. In the Smalltalk context this is not a particular problem, as Smalltalk systems are usually built out of many small objects which communicate with each other, each object consisting of only a few lines of code. All of the classes and methods nested inside a particular section can be compiled at once, if desired. Sections later in the document can be used to over-ride earlier sections, allowing patches or variant versions of the program to be included with the original code.

The document formatting capabilities are provided by the Smalltalk system's normal methods, and do not need special attention or learning by the user.

The interpreted interactive nature of the Smalltalk literate programming system is more attractive than a traditional edit, tangle, and compile cycle, mainly because the overhead for

tangling and compiling code is not necessary. Incremental tangling (and possibly weaving) of literate programs for traditional computer languages could be used to improve this situation.

Reenskaug and Skaar's experience of their literate programming system suggests that a good initial idea about what is wanted and how to do it is necessary. They suggest that literate programming is good for developing programs, because the assumptions and "soft" decisions built into the program become visible, but have reservations about its use during the initial experimentation or final testing of programs, where the specification or details of the program may be changing rapidly.

5 Language independent WEBs

The close tying of Knuth's WEB to the Pascal language may have been one reason for the lack of interest in literate programming; it has been seen, in the context of Pascal, as good for teaching but not useful for real-world programming. There have been several systems created which allow any programming language to be used, but this flexibility comes at a cost: these systems do not pretty-print the code sections, and more importantly, they do not include the index of identifiers which is so useful in WEB programs. These features have been described as *paradigmatic* by Thimbleby (in his review of Lindsay's file difference program [55]), to indicate that they are part of what makes a literate programming system *literate*. Thimbleby suggests a set of features for literate programming systems which should be essentially effortless for the programmer to use.

Hanson [39] used a simple literate programming system called `loom`, which was originally written by Incerpi and Sedgewick for use in Sedgewick's book *Algorithms*. Text and program are intermixed, with text before and after program fragments. `Loom` is implemented as a preprocessor which extracts the fragments of program and text, pushes them through a system of filters, and integrates the output. Most of the indexing, cross-referencing, and pretty-printing facilities of WEB are not provided by `loom`, but it does have filters to create an index of identifiers. Hanson's example includes replacement code introduced after profiling which generates conditional code in the output program.

Williams's FunnelWeb [93] (named after a particularly nasty antipodean spider) is a literate programming tool in the style of Knuth. Instead of two programs to extract the source code and typeset listing, FunnelWeb provides one program to perform both functions. FunnelWeb does not contain knowledge about the target programming language (in fact output files do not need to be any computer language—they are treated as free text), so it does not perform the pretty-printing and indexing of WEB. It does contain knowledge about the target formatter, but it normally neutralises formatter commands, typesetting the documentation exactly as it appears to the user. Special control sequences are provided to provide simple formatting capabilities, which can then be translated for an appropriate formatter. There is an escape mechanism to allow use of formatter control sequences if necessary. FunnelWeb forces the user to declare if a piece of code is being added to another module; this is at odds with most other literate programming tools, where the typeset listing will indicate if a module is being extended, but the input syntax is the same for all similarly named modules.

FunnelWeb includes a scripting language, which can be used for processing multiple files, writing utilities for FunnelWeb, and regression testing. The FunnelWeb shell was written so that FunnelWeb utilities did not have to be re-written in the plethora of command languages available on different systems. A suite of scripts are included with FunnelWeb to perform automated regression testing.

FunnelWeb has an option to provide a similar facility to Thimbleby's **cweb** with respect to Makefiles. If the files written out are identical to previous versions, the previous versions are left untouched, allowing minimal re-compilations to be performed easily. Williams is considering modifying FunnelWeb to read files containing pretty-printing instructions for the code parts of documents.

An alternative approach to creating literate programming tools is exemplified by `noweb` [71]. Ramsey built a minimal literate programming system, in the style of early UNIX programming tools, which supports a syntax very similar to WEB. The `notangle` and `noweave` tools convert the input syntax into an intermediate representation which is easy to manipulate using common UNIX tools. Shell scripts are used to compose the tools which convert the input syntax to intermediate representation and intermediate representation to final output. The system as implemented is simple, and does not provide pretty-printing or automatic identifier indices, but the modular construction makes it easy to insert specialised tools for particular languages where required. The `noweb` system is nearly formatter independent, having about 30 lines of AWK which convert the intermediate format for TEX or LATEX. `noweb` supports multiple programs in each source file, allowing the user to specify which root node will be expanded by `notangle`. `noweb` also provides a tool to convert the source files into a normal program (like tangling), but with the documentation included as comments, so that programs written under `noweb` can be removed from `noweb`'s control easily.

Ramsey's work is also interesting because it includes the only published case of multiple programmers using a literate programming tool to work on a single program [72]. A language-based editor (Penelope) intended to help programmers formally verify Ada programs was created using a system based on Levy's CWEB. After three years of work, the source document for the editor was over 33,000 lines long, of which about 13,000 are documentation. Seven programmers have worked on the project during that period, with up to four working concurrently. Ramsey likens a literate program that is being extended and maintained to a car repair manual, rather than the polished novel which Knuth's finished works resemble.

The WEB-based literate programming system proved inadequate in some areas; describing data structures was hampered by lack of support for diagrams or pictures; tables and figures were difficult to present; the pretty-printer's choice of line breaks and indentation were disliked. The output of WEAVE required tedious changing to include in other TEX documents, and was even more awkward to include in LATEX documents. The marked difference between the appearance of the WEB source and the typeset listing was a source of confusion. Auxiliary tools to extract parts of the WEB program were written, so that small parts of the source could be printed without the paraphernalia of large indices and tables of contents.

Ramsey concludes:

> "We cannot say to what extent literate programming can replace standard software development methodology. However, putting a clear description of design in our source code helped a changing team of programmers to develop it over a span of three years ...
>
> We believe that literate programming helped us substantially. This belief is based not on measurements but on our subjective comparisons of experience on this project to other projects. A programmer who has used standard software development systems at an international computer manufacturing company reports that a key difference in Penelope was that the documentation was *used*, precisely because of its proximity to the source code."

The VAMP system for program refinement [87, 89] has been in use and development since

1982. The documentation and code sections of literate programs are called *stubs* and *texts* in VAMP. Stubs are a set of contiguous lines starting with a start-stub command, and ending with an end-stub command. Slot commands may be included in stubs to indicate positions where other stubs are substituted during code expansion. A feature which is not found in any other literate programming system is that slot commands may contain an indication of a part of the current stub which is to be omitted if another stub is substituted for the slot. If this facility is used properly, refinements can be compilable and executable at all levels of expansion, and temporary declarations may be introduced to make stubs compilable, but replaced at a later stage of development. The VAMP system was developed on VAX/VMS, using the RUNOFF document formatter to typeset source listings.

A criticism of VAMP is that it does not insert typographic markers to indicate where slot commands are, or even which stub is which. In a complicated document with many refinements and omitted slot texts it is thus difficult to understand the actual structure of the program. VAMP is intended to be more or less formatter independent, so it is not obvious how slots and stubs could be marked. VAMP does not provide the pretty-printing or identifier indexing of `WEB`, another consequence of the goal of formatter-independence. VAMP does allow creation of more than one output file from one or more input files; the output files are not limited to programs, and can include data, tables, *etc.*

CLiP [86, 88] is a successor to van Ammers's VAMP system for literate programming. CLiP takes a unique approach to formatter and language independence which allows it to be used with almost any language and formatter or word processor. The refinement modules and references (termed stubs and slots by van Ammers) are indicated by specially distinguished programming language comments. The particular form these comments take can be tailored to the languages used by changing run-time parameters of CLiP. This approach requires the programmer to use a programming *style* rather than learn an extra set of directives to drive the literate programming tool. Also, any formatter or word processor which the programmer is familiar with can be used to create the program, reducing the learning overhead for CLiP.

Documentation sections of CLiP programs can be formatted using whatever facilities the formatter used allows, including pictures, graphs, tables, or whatever ancillary material is required. The code sections are formatted by including them in the word processor or formatter's "literal" or "verbatim" mode. The code sections of CLiP programs are not fully parsed by CLiP, so no automatic indexing of identifiers can be performed. Cross-referencing between sections can be done manually using whatever facilities the formatter or word processor provides.

CLiP can extract multiple output files from one or more input files. Top level stubs are marked with a directive giving the name of the output file. One stub is normally required to replace each slot, with the replacement done on a line-by-line basis. There are several options which can be used to change the behaviour of the stub/slot replacement mechanism. These are:

multiple This option indicates that multiple stubs may be provided for the slot in which the option appears. The stubs are concatenated together in the order in which they are found. This is similar to the behaviour of `WEB`, but `WEB` does not allow the possibility of rejecting multiple definitions for a refinement.

leader Multiple slots may have one *leader* stub defined, which will be inserted at the front of the multiple slot only if there are any other stubs provided for the slot. This facility can be used to insert required keywords in front of syntactic placeholder slots (such as constant or variable definitions).

quick Short stubs can be defined using the *quick* option, which terminates the stub at the first blank line, rather than requiring an end of stub comment.

comment off The comments which delineate stubs and slots are normally extracted with the code sections. This behaviour is undesirable in some cases, such as when data files are incorporated into the literate program. The *comment off* option prevents the stub and slot comments from being included in the output file. This option can be used locally to suppress the information about the slot in which a stub is inserted.

optional CLiP provides a mechanism in which slots can be omitted completely, for example for debugging information which is not used in production versions of the code. If a slot contains the *optional* declaration, an error will not be flagged if a corresponding stub does not appear. A separate input file can contain debugging stubs which will be included if the file is specified on CLiP's command line.

default The *default* option is useful for porting and tuning code. If a stub contains the *default* option, it will only be used if there is not a normal stub which matches the same slot. Portable versions of code can be declared in default stubs, and overridden by optimised versions in machine-specific input files, or error messages for unimplemented functions can be put in default stubs, and overridden by proper stubs as they are written. This facility allows CLiP to be used as a design tool, to create the overall plan of the program and flesh out details later.

More than one of these options can be specified in one stub or slot.

CLiP and VAMP were designed as stepwise refinement tools, rather than specifically literate programming tools, and the facilities of these tools (such as stub redefinition) are weighted to this end. (CLiP stands for *Code from Literate Programs*, emphasising its code extraction rôle.) Whilst CLiP does not provide the full facilities which might be expected of a literate programming tool, it is the only completely formatter and language independent literate programming tool to date.

6 Multilingual WEBs

Another approach to adapting WEB to different languages was taken by Ramsey's Spidery WEB system [70, 69]. In this system, the user can build specialised WEBs for different languages by creating a description of the language and processing it with the Spider tool. The class of language which Spidery WEB can handle is restricted to languages which have LR(1) grammars (Spider uses yacc to parse the scraps), and have similar lexical properties to C or Pascal. The description file contains a partial grammar of the language, and also defines translations of the language tokens for pretty printing (*e.g.*, the token != could be defined to pretty-print as $\neq$), and the token categories (which are used to alter the operator spacing to emphasise precedence). Spider provides facilities to insert compiler directives to keep track of the source code line numbers, if the language supports them.

Ramsey's Spidery WEB system has been used to build WEB systems for C, Ada, SSL, AWK, LARCH, and Reduce [34]. A version of WEB for Maple may soon be available from the authors of the Reduce WEB. The Spidery WEB system makes the need for continual adaptations of WEB to yet-another-language redundant for the most part, but does not add any new ideas to the literate programming field.

Krommes's FWEB [51, 52] is the archetypal "kitchen-sink" development of Knuth's WEB. Not only does it provide a literate programming tool for Fortran-77, it also manages to support Fortran-90, C, C++, TEX, and has a powerful enough macro processor to translate Ratfor and some M4 commands, *within the same document.* This is the only language-specific system which supports more than one language concurrently. Just about every aspect of the documentation and code layout for FWEB can be customised, including the colour of the text! FWEB input files are still structured very much like Knuth's WEB—the code is derived from Levy's version of CWEB, which in turn was derived from Knuth's WEB. The major modules of WEB which indicated important divisions between parts of the document have been expanded to include a hierarchy of possible section types.

The language pretty-printing and identifier indexing information in FWEB is built into the FWEB code, rather than being read from separate tables. This would be an obvious extension to many of the WEB systems presented so far; it is surprising that it has not been implemented yet. FWEB supports LATEX as a document formatter as well as Plain TEX; some of the facilities missing from WEB (such as documentation cross-referencing) can be thus be provided if the user is proficient in LATEX.

FWEB must count as the most capable example of a literate programming tool based on WEB. However, the future path for development of FWEB is unclear—how long new languages and customisations can continue to be added before the complexity of the program becomes overwhelming is uncertain. It is probably a testament to the power of literate programming to present problems effectively that FWEB has managed to reach its current state, with only one person driving the development.

7 WEBless WEBs

Fox [33] created a "WEBless literate programming" tool for C, called c-~~web~~ (pronounced "see no web"). The implementation of c-~~web~~ is clever; a comment at the start of the program contains an `\input` statement which TEX uses to input a macro package which implements the special formatted treatment. All c-~~web~~ commands are contained within normal C comments. The C compiler ignores the comments, but TEX reads the macro package and typesets the program according to the c-~~web~~ commands. The advantage of the method used by Fox is that the program is directly compilable using a normal C compiler, and can be formatted using TEX, with no other pre-processors. However, c-~~web~~ is in effect just a C pretty-printer, lacking any of the refinement features which make literate programming interesting.

The `doc` style file [59] for LATEX performs a similar task to c-~~web~~ for LATEX style files, allowing documentation and code to be intermingled in the same source file. Again, the refinement steps are not provided, but output to multiple files from a single source is provided. The latter facility is used to write driver files and indices which can be run through LATEX or MakeIndex as appropriate.

8 WEB tools

Several tools to facilitate literate programming have been created, but nothing as grandiose as the integrated environment or IPSEs that Knuth and Thimbleby envisaged.

An interactive interface to WEB for the GNU Emacs customisable text editor was written by Mark Motl at Texas A & M University [60]. This gives the emacs editor a limited facility for

editing and compiling WEB programs. The operations provided include movement by modules and major sections, creation and deletion of modules, and selection of modules from a simple table of contents and index. The WEB mode is driven by menus, which can be invoked by binding Emacs commands to keys.

A more sophisticated interactive literate programming system was proposed by Brown [20, 23, 22, 21], and partially implemented. The design of the literate programming environment was based around a WEB editor, with a source-level WEB debugger, a WEB importer, high-level language importer, personal preference database, and a control panel to guide the action of the other parts of the system. The principles guiding the design of the literate programming environment were that it should bring the advantages of the typeset program listing to the screen, and that it should automate many of the common tasks involved in WEB programming or provide utilities which ease significant portions of the literate programmer's work.

In practice, the former principle means that the automatically-generated information which is available from the WEB system (such as a complete index, list of named module, and module cross-reference list) should be available within the editor. The program should be displayed without WEB commands in the source interfering with the comprehension of the code. It was not thought necessary to have exactly the same display on the screen as the typeset listing provided, but it was noted that this would be best to increase the reading and comprehension speed to the same levels as the listing.

A prototype of the WEB editor at the heart of the system was built, and studies of student programmers were performed using this editor. The WEB editor included a hypertext-like interface to WEB, where the relationships between various modules were used to provide instant links from one module to another. The relationships between modules which were deemed useful were:

1. Module n uses module m. If a module includes another module, or is included by another module, a button appears on the screen allowing instant access to the other module.

2. Module n and m both appear in the same index entry. Related modules may not necessarily be direct ancestors of each other. Buttons were displayed to access other modules which were categorised under the same topic in the index.

3. Module n is an addition to module m. Modules which augment other modules were made instantly accessible from each other.

4. Modules n and m both use the same variable. This case was used to allow cross-referencing between the definitions and uses of variables. There does not appear to be any reason to restrict this sort of cross-referencing to variables rather than procedures, functions, and other symbols, but Brown mentions only variables. This may just be an oversight.

The WEB editor also includes an index display and a graphical representation of the program, which the user can browse to get an overview of the relationships between modules.

The students' performance on typical maintenance tasks was evaluated when using either the WEB editor and a normal listing of the program, or a normal text editor and a (normal) listing of the program. An experiment was performed to measure the difference in code comprehension between the methods mentioned above, using a display-only version of the WEB editor. Tasks were given to the students which involved identifying the places at which changes needed to be made to accomplish a goal, and to suggest (in English) what changes needed to be done. The results were marked on the correctness of the places which were identified, and the accuracy of the

changes suggested, as evaluated by an experienced literate programmer who had performed the same changes on the programs involved. The difference in comprehension between the methods turned out not to be statistically significant, but there was a correlation between the resources used and their performance. The students who performed best when given the standard editor and listing tended to use the listing almost exclusively. The same students, when given the WEB editor and a listing, tended to use the WEB editor in preference to the listing.

A questionnaire was given to the students in the period after the experiment had finished, in which they were asked to rate editors and the listing for ease of use and comprehension of the code. The ratings were similar for the listing and the WEB editor, but significantly worse for the standard editor. A further question asked whether the students felt they would do the exercise better with the WEB editor than the standard editor, if they had to do it again. The WEB editor was unanimously chosen.

It is not clear from Brown's thesis whether all of the features in the design plan for the WEB editor were implemented in the prototype editor. Some of the facilities which were probably omitted were:

1. Change file editing. The design for the editor included the facility to alter the literate program, and have the modifications saved as a change file, rather than as alterations in the original source. This would ease the use of changefiles significantly, and allow many programmers to work on the original source simultaneously, without interfering with each other. The changes could then be integrated into the master copy using a utility like Guntermann and Rülling's TIE, or Sewell's WEBMERGE.

2. Graphical representation of data structures.

3. Data flow diagrams. The plan for the editor included the ability to create diagrams showing how and where variables were used. This facility was intended to include aliases for variables created using WEB's macro definitions.

4. Syntax directed features. The editor was intended to incorporate some syntax-directed features for the code and documentation sections, such as template insertion. The prototype editor was syntax-directed with respect the WEB syntax, automatically handling the division into modules, control texts and indexing. This part was designed so that the user never needed to enter a WEB command directly, but experienced users could type WEB commands if desired.

The WEB editor was the only part of the literate programming environment which was implemented. The design of some of the other parts of the system was decidedly incomplete, and read like a "shopping list" of nice features, with little attention to feasibility.

The important part of Brown's work was the evaluation studies that he performed using his prototype WEB editor. These indicated that the WEB editor was a useful tool for aiding the comprehension of literate programs, and was in fact preferred to the program listing, even though the comprehension ratings were similar. Brown concludes that the full literate programming environment is worthwhile developing. The WEB editor is the part of the design which is most radically different from normal programming tools, and the other parts of the system should be minor improvements on the normal programming tools, if not particularly novel.

An interactive graphical literate programming tool called HSD (for Hierarchical Structured Document) was created by Tung [85]. HSD provides facilities similar to WEB, with an interactive graphical interface.

HSD sections can be hierarchically structured as a directed acyclic graph, rather than WEB's flat section structure (sections can also be included in more than one other section). HSD sections can contain comments, code, names of included subsections, and *define-box* and *add-to-box* commands, in any combination and order. The latter two commands are used to defer elaboration of code until later in the document.

HSD uses a Graphical Document Descriptive Language (GDDL) internally to represent the literate program, and generates the code or document by traversing the GDDL in different ways. The document is generated by a simple pre-order traversal of the GDDL, decorating the output with formatting commands where necessary. The code is generated by a two-pass algorithm which collects the deferred code together, and then traverses the graph again, outputting code with deferred code and refinements inserted at the correct places.

HSD was developed for Macintosh computers, using a commercially-available set of interface tools. Tung does not provide details of the interface to HSD, except that it uses standard Macintosh text editing facilities.

A perspective on using WEB and other literate programming tools was presented by Bart Childs at the 1992 TEX User's Group conference [25]. The literate programming tools created at Texas A & M (mostly under Childs's supervision) were reviewed; these include:

- Mark Motl's WEB-mode for GNU Emacs,
- a WEB program module size histogram creator by Mamoun Babiker,
- a WEB statistics gatherer by Mark Gaiter,
- a change file analyser,
- a WEB structure viewer by Kevin Borden, and
- a Makefile creator for literate programs by William Needels

The data from the histogram creator is interesting, suggesting that a typical WEB program has more than 90% of the modules in a program under 25 lines long, *i.e.*, one screenful of a typical terminal. This data suggests that WEB programmers break the program into chunks which can be seen all at once. Childs also presents some data to suggest that van Wyk's characterisation of literate programmers as creators of their own programming systems is changing, and there are many more users than creators now.

The Cnest and Cscope tools described by Cordes and Brown [27, 49] can be used to examine WEB programs. Cnest illustrates the location and nesting level of a current module within the overall scope of the program code. Cscope displays the program and the modules that constitute it in an easily understood form.

Brown and Czedjo proposed a hypertext representation for literate programs which transformed the hypertext linkages into relational database queries [18]. A simple prototype of this system was constructed, to show how the mapping from hypertext linkage to database query was performed.

Bishop and Gregson's LIPED system [13] is a literate program editor which is designed to let the programmer work in a format which closely resembles the final output. It is aimed at minimal hardware systems (*i.e.* IBM PCs). Five views of the program are provided; *literate program*, *code*, *contents*, *cross-reference*, and *header*. Documentation structure is confined to major and minor headings, which are inserted in the table of contents automatically. Linkage functions between views are provided, allowing the programmer to jump between corresponding parts of different

views quickly. LIPED uses the first fragment name encountered as the root fragment. Reserved words are highlighted in the compiled code, and the highlighting is suppressed between string quotes, but not in comments. The table of reserved words can be loaded dynamically, giving a rudimentary pretty-printing facility. A data structure mapper will be included as a separate view in later versions.

Smith and Samadzadeh [79, 81] describe a tool called WEBmeter for automating software complexity measurement. This tool takes data for Halstead's Software Science measures, McCabe's cyclomatic complexity number, and some measures specific to the WEB environment. Some hand calculations were also performed on small WEB programs for Yau and Collofello's design stability measurements. A large amount of data was gathered from several literate programs, but no real conclusions were formed; the work was seen as a preexperimental study for forming hypotheses about WEB complexity and stability. They make some personal observations about the use of WEB, indicating an increase in understandability noticed for large programs, and an increase in tedium for small programs.

9 Related work

The journal *Communications of the ACM* featured literate programming in Jon Bentley's *Programming Pearls* column twice in 1986 [11, 12], and then ran an infrequent column about literate programming from 1987–1990, moderated by Christopher van Wyk [30, 91]. These columns covered a variety of literate programming systems and techniques, using example programs written to solve problems set by the column editor. The programs were reviewed by guest reviewers, and these comments on their suitability for their purpose and clarity of exposition are perhaps more useful than the example programs themselves. It is only because the programs had aspirations to *literacy* that they could be reviewed in this way, and that the reviews did not turn into either marking an exercise, or a conflict of opinion about the problem's best solution.

The column stopped in 1990 because the moderator was concerned that the only people who seemed to be writing articles were implementors of literate programming systems, and that a fair conclusion to draw would be that one must write one's own system before being able to write literate programs. This conclusion is an illustration of one of the problems with literate programming; it may help create *better* programs, and thus reduce the amount of maintenance that a program requires, but the initial effort in writing a literate program is greater than the effort in writing a program using traditional methods. Users are reluctant to change to a system which requires more effort, even if the long-term gains are worthwhile.

The work of Oman and Cook [64, 65, 62] in program layout and formatting can be interpreted to provide empirical justification for research into literate programming. They performed experiments which showed that typographical re-formatting of code can significantly increase program comprehension. They call the set of principles which they used to re-format code the "book paradigm", because they presented their code listings in the format of a book, with all of the expected features of a book—pagination, table of contents, index, chapters, and sections. Typographical features were used to distinguish between externally and locally defined identifiers, procedure calls, and keywords. A key similarity with the literate programming paradigm is that the same source file was used to create the executable program, and to generate the formatted hardcopy listing.

Their research was founded in studies of programmer's behaviour when performing common maintenance tasks; Oman and Cook state [65] that:

"All programmer comprehension studies support the existence of:

1. Mental schemata or plans that guide the programmer's comprehension of code. Programmers acquire and modify these plans through experience; they are an integral part of long-term memory.
2. Chunks or meaningful units of information that programmers use to organise and remember code.
3. Beacons or highlighted semantic clues that are used to direct the review and recognition of code. Beacons are used for searching, chunking, and hypothesis checking.
4. Multiple strategies and access paths used by programmers when working with non-trivial programs. Strategies are guided by a variety of plans and conjectures depending upon individual differences, application domains, and the implementation of the code and supporting system."

Their work in typographical re-formatting of code tries to take these factors into account, to highlight beacons, and assist with multiple access paths by providing tables of contents and indices; chunks may be distinguished by insertion of blank lines.

The typographical program re-formatting of Oman and Cook incorporates some of the elements found in literate programming—generation of table of contents, cross reference tables, and index; division of the program into sections; use of proportional width fonts, boldface and italics in formatting code to distinguish different syntactic classes. In some areas they go further, by basing the program re-formatting performed on program comprehension studies, thus producing a measurable increase in program comprehensibility.

They conclude that "Good typographical formatting reflects the underlying structure of the code by providing visual clues and a variety of ways to view the code, which in turn aids maintenance activities. . . . Think about writing a book, not just a program."

Baecker and Marcus [7] used graphic design principles to make program source text more readable, understandable, and maintainable. The basic design principles from which they worked were:

1. *Typographic Vocabulary*: distinguish tokens by use of small number of appropriate type-styles.
2. *Typesetting Parameters*: adjust text size, word spacing, headline size and usage, *etc.*, to enhance readability.
3. *Page Composition*: use grids, rules, and white space to bring out program structure.
4. *Symbols and Diagrammatic Elements*: integrate appropriate symbols and diagrams to clarify essential program structure.
5. *Metatext*: augment source code with commentaries and mechanically generated supplementary text.

They developed a *visual compiler* called SEE, which processes the source code into an appropriate form for printing. Their method is similar to literate programming in that exactly the same text is used to create the printed copy of the code and the compiled executable program. The SEE compiler was built for the programming C language, and was based around the front end of the

Portable C Compiler, modified to retain comments and preprocessor declarations in the syntax tree.

Their use of the *typographical vocabulary* for program presentation is based on the function of the token. Reserved words are not emphasised by using bold face, as most pretty printers (including WEB's) do; such strong visual effects are better used for distinguishing important classes of tokens (such as global variables). Varying word spacing and kerning can also be used to visually emphasise operator precedence. In certain cases, such as the presentation of structure declarations, the SEE compiler substitutes diagrammatic markers for the original syntax, making the program structure visually accessible.

No empirical evaluation of Baecker and Marcus's methodology is presented, but some subjective evidence for its usefulness in the form of comparative examples with processed and original source code is given. Several ways in which each of the design principles above can be applied to program presentation are discussed, and alternate representations of program fragments based on the same principles are shown.

Arab [4] implemented a program re-formatter for the Pascal language which also automatically prompted for documentation, using keywords appropriate to the syntactic units which were being processed. The documentation was inserted into the program as comments at appropriate places. The re-formatting style was based on the program comprehensibility studies examined by Arab, and was customisable within certain limits. The program was intended to make presenting and documenting programs easier.

Anand [2] started from the same concepts as Knuth, but worked to a different conclusion. He argues that the name of every unit or abstraction in a program should be a functional description of the abstraction. The naming method which he suggests is applicable to the modules of a literate program. Four criteria are given for functional descriptions:

1. The details of the entity must be predictable from the description.

2. The description must convey the function of the entity, (what it does) not its logic (how it does it) or context (where it does it).

3. The description must be as concise as possible within the previous constraints. A length of 6 to 20 characters is seen as acceptable.

4. Each entity must perform one and only one specific function.

Anand shows how naming by function enhances partitioning of modules and understandability. In this system, imperative phrases should start with verbs, pointer names are adjectival phrases. Boolean variables should be named so that the phrase "How to find if *variable*" makes sense; similarly pointer variables should be named by making sense of "How to locate *pointer*" and constants and other variables should be named by making sense of "How to find *constant*". Functions and procedure declarations use "How to *name*".

Anand proposes that the identifier naming rules for programming languages are enhanced to allow non-significant whitespace to improve the functional descriptions of elements.

Mitchell [58] tried developing a literate programming methodology based on data abstraction and a formal specification language. He also tried to provide a framework for what the programmer should write and where to write it. He used the programming language Modula-2 with the specification language OBJ, and some semi-formal "glue" of his own to tie them together.

The strength of Mitchell's work is not in his approach to literate programming, but in his view of the whole program development process, which leads to the presentation of the program in literate form.

He clarifies the concepts of *design* and *specification*, introducing the notions of *internal* and *external* design. In Mitchell's world, design is a process which involves choices; specification is the result of a design process. The terms internal design and external design are used to separate the choice of mechanics for the program implementation with the choice of higher-level interfaces and functions of the program.

Mitchell also introduces a model of development space which is interesting to review in the context of literate programming. His development space has three orthogonal directions; level of detail or abstraction, level of precision, and aspect coverage.

> "During the development of a program, a programmer can say more about the program in three ways; by becoming more precise, for example, by taking something expressed in an informal language and expressing it in a formal language; by considering more aspects of the program, for example, by starting to discuss a module not previously discussed; and by adding detail, for example, by showing how the operations on an abstract data type become procedures, thereby introducing the detail of side-effects."

Mitchell formulates a set of requirements for tools to support literate programming, based on the experience of developing the "literate" program presented in his thesis. He believes that the tools for the support of literate programming should be based on the principle that things are only defined in one place, and the same definitions are carried automatically to any other place at which they are needed. This requirement can be satisfied by the macro expansion facilities of WEB.

Mitchell states that the class of tools to support the presentation of programs can be divided into two sub-classes, which present text and diagrams. He also believes that there should be tools to present the results of analysing the program, as well as tools that prepare documents containing commentary and code. The principle of providing analyses of programs should form the basis of tools that present diagrams.

Mitchell's last requirement is that tools allow the programmer to make changes to polish programs as their understanding of the problem and program grow.

He cites Peter Naur's ideas on programming as theory building, as a model of what the documentation of a program provides:

> "We can regard what the person who writes a program knows that is not in the documentation of a program as a theory of the program. Loosely speaking, a theory of a program is all the true statements about the program; the documentation of a program is a theory presentation in that it contains enough of the true statements about the program that all other true statements can be inferred from it. What the original programmer can do more easily than another programmer is apply appropriate rules of inference to derive theorems about the original program and programs related to the original program."

The problem with Mitchell's approach is that he defeats the purpose of literate programming in two major ways.

- The framework which he uses to specify what should be written where is too constrictive, and does not allow the order of exposition to be changed to suit the audience for the program.

- No support tools are provided to assist with the integration of documentation, specification, and program, or to check the program against the specification. The programmer must do everything by hand, which results in a greatly increased work-load.

Brown and Cordes have argued that the design as well as the implementation of a program should be documented in the source code [19]. They showed examples of how the design can be integrated using stepwise refinement, data-flow diagrams, and the Jackson System of Development. They believe that WEB permits the actual design to be generated in a programming language, providing a direct link between design and implementation

Brown and Cordes also compared the literate programming methodology to the use of program design languages for the design stage of the software lifecycle [17]. The requirements for program design languages were examined and contrasted with what literate programming tools can provide, with the conclusion that literate programming tools provided all of the facilities required, and had the advantage that the modules can be expanded into a fully worked program.

Parnas [67] also believed that documentation should be the primary function of programming. He described an ideal design process in which the documentation, and the consistency and completeness of the documentation, plays a major role in guiding the design. Who should write which bits of documentation was discussed, as well as which bits of documentation should be written retrospectively, if they were not written during the actual design process.

One of the objectives of the UQ2 program editor designed by Welsh *et. al.* was assistance with creating *well-documented* code [92]. The authors highlight the requirement for views of the program as code or comment dominated, even with the same program and user combination. They believe that the WEB approach is disadvantageous for three reasons:

1. Refinement addition tends to encourage global structure over nested block structure
2. Rearrangement of program fragments creates an inconsistency between the user's view of a program and its executable semantic structure
3. Re-arrangement is not well suited to exploiting hierarchical abstraction features proposed

The approach which they took was to add an optional *comment pane* into each block window. The percentage of the space used by the comment pane can be adjusted by the user, to cope with users unfamiliar with the code as well as experts. Block comments are displayed in this pane, using the normal text-editing paradigms appropriate to text areas. Re-arrangement of the code into modules was rejected for the reasons provided, and the granularity of the block comments was not adjustable. Multilingual document editing was realised using a generalisation of this method of associating text zones with blocks. The document is seen as a hierarchy of document contexts each consisting of a sequence of zones in different languages, including text.

10 Conclusions

Most of the literate programming systems and tools reviewed in this report are non-interactive. In this category there has been a divergence between the Knuth-style tools which incorporate knowledge of the target language and formatter (*e.g.*, Krommes FWEB), and "minimalist" tools which do not incorporate knowledge of the formatter and/or programming language (*e.g.*, noweb and CLiP). The proponents of the latter systems have not addressed the issues raised by Thimbleby about the literate programming *paradigm* — that certain features such as identifier

indexing and module cross referencing should come essentially for free. The former type of systems have tended towards over-complexity to allow the user to customise every detail of the system's operation.

There have been very few interactive systems, and even fewer which incorporate any form of structure-based editing into the design. A literate program is a document and a program; it has two natural views, each of which obeys its own structural rules. The author's work is concentrating on this gap; investigating how current tools for creating structure-based editors may be used or extended to build editors for literate programs.

Cordes and Brown's review of the literate programming paradigm [27] suggests some modifications to bring it up to date. Four additions are proposed; a multilevel table of contents, a graphical user interface, program debugging tools and an enhanced index. The authors have implemented prototype versions of the second and third of these additions, and are working on the other two.

They also believe that restrictions should be made on the structure of the literate program and on the size of the command set for driving the literate programming tool. Each module in the program should function as a logical single entity, within the syntax rules for the language. This assists the reader in understanding module functionality and tracking the program scope. The second restriction removes some of the individual customisability of the output listing, but a template-based documentation generation routine is envisaged which could tailor the output for particular house styles. These restrictions would be easily implementable in a structure-based literate programming editor.

Finally, there is a new book by Knuth [48] on literate programming. While there is very little new material in this book, it does bring together Knuth's thoughts on the topic of literate program, with some of the earlier papers which led to the development of literate programming, and some examples of what Knuth's literate programming has achieved. Hopefully, this book will stimulate interest in literate programming again.

The following references were not available while this report was being written:

WEBs for other languages Ben-Ari's Foreet tool for Fortran [10], Cunningham and Beck's example of SmallTalk literate programming [28], and Detig's MWEB system for Modula-2 [31]

WEB tools Beck and Cunningham's literate program browser [9], and Sametinger's hypertext system for literate C++ programming [74].

Related work Bunge and Cisneros's literate programming for software chemistry tools [24], Childs and McGuire's symbolic computing and automatic programming with literate programming [26], Guntermann's modular programming with WEB [37], and Hines's review of literate programming [40].

References

[1] Paul W. Abrahams. **Typographical Extensions for Programming Languages: Breaking out of the ASCII Straitjacket.** *ACM SIGPLAN Notices*, Vol. 28, No. 2, pp. 61–68, February 1993.

[2] N. Anand. **Clarify Function!** *ACM SIGPLAN Notices*, Vol. 23, No. 6, pp. 69–79, June 1988.

[3] W. Appelt and K. Horn. **Multiple changefiles in WEB.** *TUGboat*, Vol. 7, No. 1, p. 20, March 1986.

[4] Mouloud Arab. **Enhancing Program Comprehension: FORMATTING and DOCUMENTING.** *ACM SIGPLAN Notices*, Vol. 27, No. 2, pp. 37–46, February 1992.

[5] A. Avenarius, S. Oppermann, I. Peides, and J. Stritzinger. **The FWEB System of Structured Documentation.** Report PI-R6/89, Institut für Praktische Informatik, Technische Hochschule Darmstadt, 1989.

[6] Adrain Avenarius and Siegfried Opperman. **FWEB: A Literate Programming System for Fortran8x.** *ACM SIGPLAN Notices*, Vol. 25, No. 1, pp. 52–58, January 1990.

[7] Ronald Baecker and Aaron Marcus. **Design Principles for the Enhanced Presentation of Computer Program Source Text.** In *Proceedings CHI'86 (Human Factors in Computing Systems)*, pp. 51–58, New York, NY, April 1986. Association for Computing Machinery.

[8] Ronald Baecker and Aaron Marcus. **Human Factors and Typography for More Readable Programs.** Addison-Wesley, Reading, MA, 1990. ISBN 0-201-10745-7.

[9] Kent Beck and Ward Cunningham. **The Literate Program Browser.** Technical Report CR-8-52, Computer Research Laboratory, Textronic, Incorporated, 1986.

[10] Mordechai Ben-Ari. **Foreet: A tool for design and documentation of Fortran programs.** *Software – Practice and Experience*, Vol. 16, No. 10, pp. 915–924, October 1986.

[11] Jon L. Bentley. **Programming Pearls—Literate Programming.** *Communications of the ACM*, Vol. 29, No. 5, 1986.

[12] Jon L. Bentley and Donald E. Knuth. **Programming Pearls.** *Communications of the ACM*, Vol. 29, No. 6, 1986.

[13] Judy M. Bishop and Kevin M. Gregson. **Literate Programming and the LIPED Environment.** *Structured Programming*, No. 1, pp. 23–34, 1992.

[14] Peter Breitenlohner. **Still another aspect of multiple change files: The PATCH processor.** *TUGboat*, Vol. 9, No. 1, pp. 11–12, April 1988.

[15] Frederick P. Brooks Jr. **The mythical man-month.** Addison-Wesley, 1982.

[16] Peter Brössler, Andreas Freiherr, T. Kampfmann, R.-D. Nausester, and Helmut Waldschmidt. **Modulare Programmierung in WEB.** Report TI-3/86, Institut für Theoretische Informatik, Technische Hochschule Darmstadt, June 1986.

[17] M. Brown and D. Cordes. **A literate programming design language.** In *COMPEURO'90, Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering*, pp. 548–549, Tel Aviv, Israel, May 1990.

[18] M. Brown and B. Czejdo. **A Hypertext for Literate Programming.** In Selim G. Akl et. al. (editor), *Advances in Computing and Information, Proceedings of International Conference ICCI'90*, pp. 250–259, Niagara Falls, Ontario, Canada, May 1990. Lecture Notes in Computer Science **468**.

[19] Marcus Brown and David Cordes. **Literate programming applied to conventional software design.** *Structured Programming*, No. 11, pp. 85–98, 1990.

[20] Marcus E. Brown. **An Interactive Environment for Literate Programming.** PhD thesis, Texas A&M University, College Station, TX, August 1988.

[21] Marcus E. Brown. **The Literate Programming Tool.** Technical report, Computer Science Department, Texas A&M University, August 1988.

[22] Marcus E. Brown and Bart Childs. **An Interactive Tool for Literate Programming,** (unpublished). in *Third Workshop on Empirical Studies of Programmers*, unpublished proceedings, Austin, Texas, April 1989.

[23] Marcus E. Brown and Bart Childs. **An Interactive Environment for Literate Programming.** *Structured Programming*, Vol. 11, No. 1, pp. 11–25, 1990.

[24] Carlos F. Bunge and Gerardo Cisneros. **Modular libraries and literate programming in software for *ab initio* atomic and molecular electronic structure calculations.** *Computers & Chemistry*, No. 12, pp. 85–89, 1988.

[25] Bart Childs. **Literate programming, a practitioner's view.** *TUGboat*, Vol. 13, No. 3, pp. 261–268, 1992.

[26] Bart Childs and Timothy Jay McGuire. **Symbolic computing, automatic programming, and literate programming.** In *Computational Techniques and Applications, Proceedings of CTAC-91*, Adelaide, Australia, July 1991. Australian National University.

[27] David Cordes and Marcus Brown. **The literate-programming paradigm.** *Computer*, Vol. 24, No. 6, pp. 52–61, June 1991.

[28] Ward Cunningham and Kent Beck. **Scroll Controller Explained, An Example of Literate Programming in Smalltalk.** Technical Report CR-86-53, Computer Research Laboratory, Textronix Incorporated, 1986.

[29] R. M. Damerell. **Error detecting changes to Tangle.** *TUGboat*, Vol. 1, No. 7, pp. 22–24, 1986.

[30] Peter J. Denning. **Announcing Literate Programming.** *Communications of the ACM*, Vol. 30, No. 7, p. 593, July 1987.

[31] Christine Detig and Joachim Schrod. **MWEB, A WEB system for Modula-2.** Report PI-R10/88, Institut für Praktische Informatik, Technische Hochschule Darmstadt, March 1988.

[32] Angus J. C. Duggan. **Literate Programming: Past, Present, and Future.** Colloquium given at National Physical Laboratories, Teddington, London, 18th November 1992. Material to appear as University of Edinburgh CS Departmental Report.

[33] Jim Fox. **Webless literate programming.** *TUGboat*, Vol. 11, No. 4, pp. 511–513, 1990.

[34] Peter Gragert and Marcel Roelofs. **Reduce WEB version 3.4,** (unpublished). Available by anonymous ftp from `utmf0.math.utwente.nl`.

[35] Klaus Guntermann and Wolfgang Rülling. **Another approach to multiple changefiles.** *TUGboat*, Vol. 7, No. 3, p. 134, October 1986.

[36] Klaus Guntermann and Joachim Schrod. **WEB adapted to C.** *TUGboat*, Vol. 7, No. 3, pp. 134–137, October 1986.

[37] Klaus Guntermann and Helmut Waldschmidt. **Modular programming with WEB.** *Electrosoft*, Vol. 1, No. 1, pp. 27–43, March 1990.

[38] Eric Hamilton. **Literate Programming—Expanding Generalized Regular Expressions.** *Communications of the ACM*, Vol. 31, No. 12, pp. 1376–1385, December 1988.

[39] David R. Hanson. **Literate Programming—Printing Common Words.** *Communications of the ACM*, Vol. 30, No. 7, pp. 594–599, July 1987.

[40] Michael Hines. **Literate Programming.** unpublished technical report, Department of Physiology, Duke University Medical Centre, Durham, NC, June 1986.

[41] Marco S. Hyman. **Literate C++.** *Computer Language*, Vol. 7, No. 7, pp. 67–79, July 1990.

[42] Michael A. Jackson. **Literate Programming—Processing Transactions.** *Communications of the ACM*, Vol. 30, No. 12, pp. 1000–1010, December 1987.

[43] Donald Knuth. **Remarks to Celebrate the Publication of Computers and Typesetting.** *TUGboat*, Vol. 7, No. 2, pp. 95–99, 1986.

[44] Donald E. Knuth. **The WEB System of Structured Documentation.** Stanford Computer Science Report CS980, Stanford University, Stanford, CA, September 1983.

[45] Donald E. Knuth. **Literate Programming.** *The Computer Journal*, Vol. 27, No. 2, pp. 97–111, 1984.

[46] Donald E. Knuth. **Computers and Typesetting, Volume B, TEX: The Program**, volume B of *Computers & Typesetting*. Addison-Wesley, Reading, MA, 1986. ISBN 0-201-13437-3.

[47] Donald E. Knuth. **Computers and Typesetting, Volume D,** METAFONT**: The Program**, volume D of *Computers & Typesetting*. Addison-Wesley, Reading, MA, 1986. ISBN 0-201-13438-1.

[48] Donald E. Knuth. **Literate Programming.** Center for the Study of Language and Information, Stanford University, CA: Distributed by Univ. of Chicago Press., 1992. ISBN 0-937073-80-6.

[49] Eduardo Kortright. **CNEST and CSCOPE,** (unpublished). available for anonymous ftp from `utah.math.edu` in directory `pub/tex/pub/cscope`.

[50] Heinz Kredel. **Software development for computer algebra; or, From ALDES/SAC-2 to WEB/Modula-2.** In P. Gianni (editor), *Symbolic and Algebraic Computation, Proceedings of International Symposium ISSAC'88*, pp. 447–455, Rome, Italy, July 1988. Lecture Notes in Computer Science **358**.

[51] John Krommes. **FWEB (Krommes) vs. FWEB (Avenarius and Oppermann).** *TEXhax*, Vol. 90, No. 19, February 1990.

[52] John A. Krommes. **FWEB User's Manual.** Princeton University, April 1992. This is an incomplete draft of a forthcoming laboratory report.

[53] O. Lecarme. **Literate Programming.** *ACM Computing Reviews*, Vol. 26, No. 1, p. 75, January 1985.

[54] Silvio Levy. **WEB Adapted to C, Another Approach.** *TUGboat*, Vol. 8, No. 1, pp. 12–13, 1987.

[55] Donald C. Lindsay. **Literate Programming—A File Difference Program.** *Communications of the ACM*, Vol. 32, No. 6, pp. 740–755, June 1989.

[56] Charles Lins. **A First Look at Literate Programming.** *Structured Programming*, Vol. 10, No. 1, pp. 60–62, 1989.

[57] Charles Lins. **An Introduction to Literate Programming.** *Structured Programming*, Vol. 10, No. 2, pp. 107–112, 1989.

[58] Richard Mitchell. **Literate Programming.** PhD thesis, Hatfield Polytechnic, U.K., February 1988.

[59] Frank Mittelbach. **The doc-Option.** *TUGboat*, Vol. 10, No. 2, pp. 245–273, 1989.

[60] Mark B. Motl. **A Literate Programming Environment Based on an Extensible Editor.** PhD thesis, Texas A & M University, College Station, TX, December 1990.

[61] Paul W. Oman. **A Taxonomic Analysis of Typographic Programming Style.** PhD thesis, Oregon State University, Corvallis, OR, December 1988.

[62] Paul W. Oman and Curtis Cook. **A Paradigm for Programming Style Research.** *ACM SIGPLAN Notices*, Vol. 23, No. 12, pp. 69–78, December 1988.

[63] Paul W. Oman and Curtis Cook. **A Programming Style Taxonomy.** Technical Report 88-60-20, Oregon State University, Corvallis, OR, 1988.

[64] Paul W. Oman and Curtis Cook. **The Book Paradigm for Improved Maintenance.** *IEEE Software*, Vol. 7, No. 1, pp. 39–45, January 1990.

[65] Paul W. Oman and Curtis Cook. **Typographic Style is More than Cosmetic.** *Communications of the ACM*, Vol. 33, No. 5, pp. 506–520, May 1990.

[66] T. L. Pappas. **Literate Programming for Reusability: A Queue Package Example.** In *Proceedings of the Eighth Annual Conference on Ada Technology*, pp. 500–514, Atlanta, Georgia, March 1990. ANCOST, Inc. (Sponsor).

[67] David Lorge Parnas and Paul C. Clements. **A Rational Design Process: How and Why to Fake it.** *IEEE Transactions on Software Engineering*, Vol. 12, No. 2, pp. 251–257, February 1986.

[68] J. D. Ramsdell. **SchemeTEX—Simple Support for Literate Programming.** *TEXhax*, Vol. 88, No. 39, April 1988.

[69] Norman Ramsey. **The Spidery WEB System of Structured Documentation.** Technical report, Dept. of Computer Science, Princeton University, August 1989.

[70] Norman Ramsey. **Weaving a Language-Independent WEB.** *Communications of the ACM*, Vol. 32, No. 9, pp. 1051–1055, September 1989.

[71] Norman Ramsey. **Literate-Programming Tools Need Not Be Complex.** (Technical Report) CS-TR-251-91, Princeton University, Department of Computer Science, Princeton University, 35 Olden Street, Princeton, New Jersey 08544, August 1992.

[72] Norman Ramsey and Carla Marceau. **Literate Programming on a Team Project.** *Software – Practice and Experience*, Vol. 21, No. 7, pp. 677–683, July 1991.

[73] Yves Roy. **TEX/WEB et le traitment de textes mathématiques.** Masson Editeur, Paris, 1984. ISBN 2-22-580249-1.

[74] J. Sametinger. **A Hypertext System for Literate C++ Programming.** *Journal of Object-Oriented Programming*, Vol. 4, No. 8, pp. 24–29, January 1992.

[75] Joachim Schrod. **The CWEB System.** Report TI-2/90, Technische Hochschule Darmstadt, January 1990.

[76] E. Wayne Sewell. **How to MANGLE your software: the WEB system for Modula-2.** *TUGboat*, Vol. 8, No. 2, pp. 118–122, July 1987.

[77] E. Wayne Sewell. **Multiple Changefiles: The Adventure Continues.** *TUGboat*, Vol. 8, No. 2, pp. 117–118, July 1987.

[78] E. Wayne Sewell. **Weaving a Program: Literate Programming in WEB.** Van Nostrand Reinhold, 1989. ISBN 0-442-31946-0.

[79] L. M. C. Smith. **Measuring Complexity and Stability of WEB Programs.** Master's thesis, Dept. of Computer Science, Oklahoma State University, August 1990.

[80] Lisa M. C. Smith and Mansur H. Samadzadeh. **An Annotated Bibliography of Literate Programming.** *ACM SIGPLAN Notices*, Vol. 26, No. 1, pp. 14–20, January 1991.

[81] Lisa M. C. Smith and Mansur H. Samadzadeh. **Measuring Complexity and Stability of WEB Programs.** *Structured Programming*, Vol. 13, No. 1, pp. 35–50, January 1992.

[82] H. Thimbleby. **Experiences of 'Literate Programming' using cweb (a variant of Knuth's WEB).** *The Computer Journal*, Vol. 29, No. 3, pp. 201–211, 1986.

[83] Harold. W. Thimbleby. **Literate Programming in C.** (technical report), University of York, 1984.

[84] Edward J. Thomas and Paul W. Oman. **A Bibliography of Programming Style.** *ACM SIGPLAN Notices*, Vol. 25, No. 2, pp. 7–16, February 1990.

[85] Sho-Huan Tung. **A Structured Method for Literate Programming.** *Structured Programming*, Vol. 10, No. 2, pp. 113–120, 1989.

[86] Eric W. van Ammers. **CLiP, a universal literate programming tool.** Announcement to LITPROG@edu.SHSU mailing list, February 1993.

[87] Eric W. van Ammers and Mark R. Kramer. **VAMP: a Tool for Literate Programming Independent of Programming Language and Formatter.** In *Proceedings of the 6th Annual European Computer Conference, CompEuro '92*, pp. 317–376, 1992.

[88] Eric W. van Ammers and Mark R. Kramer. **The CLiP Style of Literate Programming,** (unpublished). Unpublished report from Wageningen Agricultural University, The Netherlands, February 1993.

[89] E. W. van Ammers et. al. **VAMP: A Tool for Programming by Stepwise Refinement.** Internal report, Agricultural University of Wageningen, Computer Science Department, 1984.

[90] P. N. van den Bosch. **Weaving a Program: Literate Programming in WEB.** *ACM Computing Reviews*, Vol. 31, No. 7, pp. 343–344, July 1990.

[91] Christopher J. van Wyk (Moderator). **Literate Programming.** *Communications of the ACM*, Vol. 30(7), 31(12), 32(6), 32(9), 33(3), 1987–1990.

[92] Jim Welsh, Brad Broom, and Derek Kiong. **A Design Rationale for a Language-based Editor.** *Software – Practice and Experience*, Vol. 21, No. 9, pp. 923–948, September 1991.

[93] Ross N. Williams. **FunnelWeb User's Manual,** May 1992.

[94] Y. C. Wu and T. P. Baker. **A Source Code Documentation System for Ada.** *ACM Ada Letters*, Vol. 9, No. 5, pp. 84–88, 1989.

[95] Kongshi Xu. **Report on R and D at the Software Institute.** In G. X. Ritter (editor), *Information Processing 89, Proceedings of the IFIP 11th World Computer Congress*, pp. 73–76, San Francisco, CA, August-September 1989. Mentions Yunmei Dong's CDS system of literate programming, combining C with a language called SP for typesetting Chinese and English.