Experiences of 'Literate Programming' using cweb (a variant of Knuth's WEB)

H. THIMBLEBY
Department of Computer Science, University of York, Heslington, York YO1 5DD, United Kingdom

Cweb ***is a literate programming system for the programming language*** **C.** ***Experience developing and using it forms the basis of this paper, the purpose of which is to:***

- ***support Knuth's enthusiasm for literate programming;***
- ***discuss developments in literate programming support, both within the current framework of*** **cweb** ***and to interactive graphics support;***
- ***discuss the implementation issues, considering*** **cweb** ***as part of a simple IPSE (Integrated Project Support Environment); the conclusions for IPSEs appear pessimistic.***

Received November 1984

1. INTRODUCTION

After hearing Donald Knuth extol his literate programming system, WEB[7,8], I decided to implement a UNIX* version of it, which came to be called **cweb**. **Cweb** is a tool to facilitate high-quality program documentation in a combination of **C** (the programming language) and **troff** (a text-formatting language).

In principle, literate programming is not language dependent; it is a system for annotating and decomposing formulae of various kinds so that the formulae are recoverable. Means are also provided to format the combined annotations and formulae to a high standard of presentation and to provide derived cross-reference information. The idea has a wide range of application:

- for commentaries on classical literature, e.g. on Virgil's *Aeneid*;
- for multi-lingual commentaries on computer programs – see the Japanese translation of Knuth's paper[8] by Toshiaki Kurokawa[9];
- for formal commentary on programs, e.g. combining programs with their specification;
- for informal commentary on programs, e.g. books about programming;
- for annotating a formal record of an interactive session – see section 7.1 below.

In **cweb** the formulae are **C** programs and the annotation is free natural language text; **cweb** is thus a notation to associate **C** program code and its documentation as closely as possible, and a system which can convert this interleaved description into either program source or to documentation annotating the source. The source code generated is, of course, directly acceptable to the **C** compiler. The combined code and documentation can be processed and possibly typeset to result in a high-quality presentation including a table of contents, index, cross-referencing information, and due regard for typographical conventions (such as bold fonts for reserved words). The result is called a 'literate program' because the final document is not only readable, but may actually be appreciated as literature *for* humans. Literate programming changes the programmer's perspective from programming for a machine to explaining to other people (peers, students or examiners!) what he intends the machine to do. The main concern of the programmer, hopefully, becomes excellence of style and exposition. To help make the exposition clearer, literate programming systems enable the program to be written down in an order and with a structure (or 'web') that is most comprehensible rather than strictly obeying the topological laws of the underlying programing language. In fact, the web mechanism is a static alternative to procedural abstraction in the programming language which enables fragments of code and documentation to be abstracted and related. **Cweb** differs from Knuth's WEB system mainly in the choice of languages: WEB is based on Pascal and TEX rather than **C** and **troff**/**nroff**. In addition WEB provides a few extensions to Pascal (e.g. a more flexible identifier syntax) which are partly or fully available in standard **C**.

After I had been working on **cweb** for a short time, Knuth sent me the literate program for his system. Although it was written in an awkward subset of Pascal (in order to be portable) and running into 173 pages, I actually enjoyed reading it and I felt had a good grasp of its inner workings after only a few hours. I wanted to see if I could achieve the same results with **C**.

Cweb was designed on the following principles.

(1) It should provide simple but sufficient facilities for literate programming but should not enforce particular styles of programming. In particular, any previously valid **C** program text should be handled correctly.

(2) It should permit the programmer maximum control over the format of the documentation but should provide sensible defaults so that it could be used to the full without the programmer 'going into' formatting as an end in itself. It is assumed that **C** programmers are probably prejudiced in favour of a constant-width font (such as used in Kernighan & Ritchie[6]), but an Algol publication-like presentation (italic identifiers, strings with explicit spaces, bold reserved words...) should not be precluded. A discussion of the presentation of source code can be found in Baecker and Marcus.[1]

(3) There should be few automatic features: the programmer should get exactly what he asks for. This is to facilitate programmers experimenting with styles of literate programming and requiring explicit control over the system. All of the automatic features actually provided by **cweb** are orthogonal and can be controlled independently.

* UNIX is a trademark of AT&T Bell Laboratories.

(4) Finally, although it was recognised that **cweb** would be a transitional tool in the sense that what was learned from its use would inevitably lead to its eventual obsolescence, it should be implemented thoroughly. The interesting design issues in **cweb** are related to human factors and details of 'feature integration' – the actual implementation is relatively trivial. In design one learns only to the extent that attention is paid to detail. Thus **cweb** is a full production implementation, in the hope of exposing as many design issues as possible.

As usual, the original aim for elegance was compromised in order to achieve sufficiency without interfering with existing practice. It should have been no surprise to rediscover the system design rule

Generality increases sublinearly and complexity increases superlinearly when combining unrelated systems.

This paper reports on my experience of trying to implement a good idea in what turned out to be a most tortuous environment – **C, troff** and UNIX. **Cweb** barely qualifies as an Integrated Project Support Environment (IPSE) but it is surely a plausible part of an IPSE, in the sense that an IPSE might hope to integrate documentation and program code, so I hold out even less hope of success for more ambitious attempts at IPSEs that integrate existing tools (such as general-purpose programming languages, databases and so on). Notwithstanding these difficulties, literate programming deserves greater attention, and later parts of the paper explore possible developments.

2. HOW CWEB WORKS

2.1. Basic notation

For explanatory purposes it is easiest to think of **cweb** as a conventional macro processor. Given a file of interleaved documentation, macro definitions and calls, **cweb** can be asked to evaluate the macro calls (and produce source code for a compilable program) or to translate the same specification into input to a text formatter so that the program and associated documentation are presented as usefully as possible.

A macro definition has the form:

@< *name* @>= *body*

body is terminated implicitly at the start of the next section of the web specification (*q.v.*). The same syntactic form may be used subsequently to append to the body. The choice of '@' and the symbols '@<', '@>=' is arbitrary: merely that they do not occur naturally in **C**.

A macro call has the form:

@< *name* @>

When code is produced, each call is textually replaced by the corresponding body, or code bodies concatenated in their original order if there was a series of definitions. To make **cweb** easier to use, macro calls may occur before and after definitions, without regard for definition before use. Macro names are arbitrary length and may be abbreviated by appending '....'. Abbreviation may be used before and after a full occurrence of a name. Note that the abbreviation mechanism is decoupled from the definition mechanism, in the sense that definitions of macros need not use their full names. In addition, white space (sequences of spaces, newlines, tabs) in names compare equal, so **cweb** is relaxed about the layout of names. As will be seen shortly, names may contain embedded **C** code for expository purposes.

Documentary structure is introduced by defining macros in units called 'sections'. A section has two parts, documentation and code in that order, and may take one of two forms:

@	@
documentation	*documentation*
@< *name* @>=	@c
code	*code*

In either case, *documentation* or *code* parts may be omitted. The section on the right is 'unnamed', its code is appended to the initial text which forms the basis of the source code: the code of an unnamed section is treated as if appended to the body of a single anonymous macro, the body of which is actually the source code corresponding to the web specification.

The **C** language allows programs to utilise source code from existing files (using the '#include' mechanism), frequently to gain access to library or other definitions. **Cweb** has an orthogonal feature whereby macro bodies can be placed in named files which can then be re-used with an 'include' directive. This allows, for example, global definitions to be written as near to their associated operations as wished (e.g. for documentation purposes), but for the actual code to have more extensive scope because, once placed in a separte file, it can be included many times over elsewhere. To achieve this, a macro name may be given a quoted prefix in the form '"*file-name*"'. The body of the macro is then written on to the named file. This is the effect, but the implementation of the write is arranged to minimise actual changes to the file and to permit several macro bodies from diverse **cweb** source files to accumulate within the same file. It works as follows. **Cweb** searches the named file for macro bodies from the current **cweb** source file (the one defining the macro). **Cweb** will then only update the file if the macro body is new or changed. All other text in the file is left unaffected. Finally, this form of macro otherwise behaves normally, and it may be appended or invoked as any other.

So far in the explanation, **cweb** really provides no more than a convenient macro processor with features for commentary and features which facilitate restructuring the program with respect to the order and distribution of definitions and calls. The second function of **cweb** is to take *the same* specification and present it aesthetically, as a work of literate programming complete with the trimmings of a significant work of literature: table of contents, cross-references and indices. **Cweb** processes the same specification and produces code to drive a text formatter, which in turn can drive a line printer, phototypesetter or other hard-copy device. In this mode **cweb** therefore takes full account of font details, such as italics for variables, bold for reserved words and so on.

Two main extra features for hard-copy presentation of the literate program are: first, a heading to introduce a major section of a program

@* *heading*
documentation
...

This *heading* will later appear in the table of contents. And second, a notation for flagging code in documentation, comments and macro names, so that it may be

formatted using the same typographical conventions as code elsewhere. Thus in documentation any text between dollar signs is formatted with the same conventions as other program code, but it does not contribute to the executable program code. (Again, '$' is an arbitrary symbol.) Any text formatted as code, whether or not ultimately executable, is automatically scanned for identifier uses to make index entries.

2.2. Small example

Given the **cweb** specification fragment:

specification

```
@* Maths Functions
We define a function $exp(x, y)$
to return $x$ to the power $y$.
@c
long exp(x, y)
{ long z;
  @<Check $y$ is positive@>
  ...
  return z;
}
@ @<Check $y...@>=
  if( y < 0 )
    error ...
```

Troff is capable of formatting x^y, so one could achieve 'We define a function...to return x^y' using suitable commands, but the fancy superscript notation would distract from this example. As it stands the example would produce the following output when run through **cweb**, then formatted by **troff**:

formatted output

12. **Maths Functions.** We define a function *exp*(*x*, *y*) to return *x* to the power *y*.

```
long exp(x, y)
{       long z;
        ⟨Check y is positive 13⟩
        ...
        return z;
}
```

13. ⟨Check *y* is positive **13**⟩ ≡

```
if (y < 0)
  error...
```

This code is used in section 12.

The following C code might be produced from the same specification. The lines starting '`#line`' are compiler directives added by **cweb** so that any compiler diagnostics would refer back to the correct line (and section number) in the original **cweb** source file, which we assume here was called 'mathlib'. Unfortunately the diagnostics will still be rather cryptic (what is 'sec. 13 in 12 mathlib'?) because the compiler cannot handle longer strings in this context: the line-numbering compiler directives were originally intended for more modest applications.

code output

```
#line 237 "sec.12 mathlib"
long exp(x, y)
{ long z;
#line 244 "sec.13 in 12 mathlib"
if( y < 0 )
error ...
#line 240 "sec.12 mathlib"
...
return z;
}
```

2.3. Additional features for controlling format

Unlike WEB, **cweb** does not pretty print by changing the layout of programs (e.g. using an algorithm such as the one presented by Oppen[11]). In **cweb** it is assumed that the programmer has access to a display editor, and the program layout he chooses is the preferred one anyway. (Actually, **troff** is not as general as TEX, and pretty printing cannot be specified in terms of device parameters such as line length.) But the fonts used in formatting need not be constant pitch, and the layout the programmer wants may not be retained exactly in the formatted version. **Cweb** provides 'alignment' markers (or 'self-setting' tabs) such that all *n*th alignment markers '`@`*n*' within a particular section are aligned vertically when finally formatted. An example:

specification

```
#define SPACE@1 ' '
#define TAB@1 '\t'
#define NEWLINE@1 '\n'
...
case SPACE:
case TAB:
case NEWLINE:@3 continue;
case '-':
case '+':@3 sign = c;@4 break;
case '.':@3 dot = true;@4 break;
default:@3 @<error@>
```

formatted output

```
# define SPACE    ' '
# define TAB      '\t'
# define NEWLINE  '\n'
...
case SPACE:
case TAB:
case NEWLINE: continue
case '-':
case '+':       sign = c;     break;
case '.':       dot = true;   break;
default:        ⟨error 42⟩
```

Here, all the code to the right of the colons is aligned despite the varying widths of the **case** constants, and the two **break**s are vertically aligned over each other. There need be no horizontal relationship between the different numbered alignment columns, except that in this case '`@4`' is somewhat to the right of '`@3`'. This alignment mechanism is an obvious candidate for automatic specification (which could easily be done using an attributed grammar to place the '`@`*n*'), but by adherence to the present design principles this has not been attempted. Additional explicit control of fine spacing is provided by codes which insert small spaces and by a 'literal' notation which permits the embedding of arbitrary **troff** control code within **C** text.

As **troff** is rather basic, **cweb** also provides full control over font correction, to avoid certain printed characters being awkwardly spaced or even colliding when, say, the ascender of an italic letter (like '*f*') leans towards an upright character (like a bar, '|').

The few remaining features, which need not be discussed in detail here, assist the programmer in constructing cross-references, controlling fonts, contributing to a more helpful index, and so on.

3. THE RELATION OF CWEB TO MODULAR PROGRAMMING

A macro-processing isomorphism was used to explain the operation of **cweb** above. Equivalently it is possible to view **cweb** as introducing a notation for statically invoked procedures with dynamically bound free identifiers. Intentionally, **cweb** does not restrict the procedure body to 'sensible' or even well-formed categories. Dynamic binding has often been criticised on the basis that it reduces program clarity (e.g. Tennent[16]), particularly because all local identifiers are implicit object parameters to procedure invocations. Thus local or private information may be subject to unintentional corruption in other components of a program. Also for a language which uses dynamic binding for run-time invocations, this form of privacy invasion cannot be detected statically in general. **Cweb** does not suffer so seriously, for two reasons. First, a distinctive notation is used for invocation of procedures with dynamically bound free identifiers. Secondly, the invocation is static, and so type and other security checking can be performed statically.

But dynamic binding provides a significant benefit which is an especially important contribution to structuring developing programs: the hierarchal structure of the invocations of these procedures is semantically transparent. Thus the programmer is free to restructure the program without transforming its text in any other way. This is very important if the program is being documented, or the existing documentation is being refined. New, clearer ways if structuring the program may occur to a programmer as he documents, and he can make quite radical changes – moving code between **cweb** procedure/macro definitions and their invocations – without risking modifications to the existing code.

For these reasons it is not anomalous that **cweb** procedures do not have explicit parameters. If they had, changing an explanatory abstraction and its documentation might entail changing code, which might not be achieved without introducing errors. However, in Marneffe and Ribben's related 'Holon Programming' system[10], the so-called holons (which correspond to our macros) could have explicit parameters. Then after analysing the web and flow of control, their system chose open or closed replacement for invocations.

The special notation used for a procedure invocation permits an arbitrary length and much more explicit name for a module than would be possible within the identifier micro-syntax of the programming language. The name-abbreviation mechanism is to encourage the use of sensible names which might otherwise be far too tedious to type in full at each occurrence in the base programming language. Indeed, names (being treated as documentation) may refer to identifiers and language constructs explicitly without difficulty, using the '**$**' notation.

4. THE BENEFITS OF LITERATE PROGRAMMING

The initial impression of **cweb** is a hacker's paradise, emphasised by the use of brief messy keywords like '`@*`', but experience with **cweb** does not bear this out.

Cweb allows the programmer to interleave documentation and program with considerable ease. The important point is that the (trivial) notation to support this is unobtrusive: if documentation (or even commentary) is going to be written anyway, the 'keystroke' overhead of using **cweb** is negligible.

Here are more points in **cweb**'s favour:

- The effect of expounding about a program (as in lecturing) is to point up deficiencies. Any omissions become apparent and program quality improves. **Cweb** also motivates by making documentation pleasing to the eye, and by providing indices and cross-references. **Cweb** provides much more incentive to 'document-as-you-code'.
- When documentation and program are as closely coupled as they are in **cweb** files, they are very much more likely to be consistent with each other. When programs are developed on-line, or even typed-up from paper notes, the ease of dropping into documentation encourages the programmer to express his intentions fully. If a source file is edited, perhaps to refine a program fragment, the relevant documentation is readily brought up to date at the same time. Thus **cweb** encourages both full and consistent documentation.
- Because **cweb** permits full access to **troff**, documentation can make full use of the typographical capabilities that come with **troff** and its various support tools for setting equations, tables and diagrams. In particular, mathematical statements can be formatted using proper symbols. Less obviously, **cweb** can also be used to make stylistic adjustments to the program, for example to cause a **C** identifier name 'NIL' to be printed as the greek letter 'ϕ' throughout. (The same tricks can be used to compensate for grotty line printers which cannot distinguish letter '`l`' from digit '`1`'.)
- A language such as **C** requires declarations (and other syntactic categories) in particular order, and external functions not returning integer values require declaration before use, etc. **Cweb** largely overcomes this problem. Also, definitions (including **C** macro definitions) which are required in common header files

may be declared close to their use, and **cweb** will maintain the header files.

- **Cweb** is particularly useful in the implementation stages when following a system design methodology; it automates the construction of a linear **C** program from an arbitrarily nested abstract specification (the web). If the system structure has not been fully refined, **cweb** is able to produce an executable **C** program for testing and development purposes: if undefined modules are used, **cweb** substitutes a call to a **C** procedure which can be used as an executable stub. If the system structure is ill formed (e.g. recursive), **cweb** produces helpful documentation with appropriate diagnostics.
- If one writes a routine to perform some task *T*, by the time error checking, diagnostic generation and error recovery have been included the routine to do *T* appears to be an error routine, with *T* occupying maybe 5% of it somewhere at the bottom. So a good programmer omits or curtails the checking to ensure that the routine *looks* as if it does *T*. **Cweb** provides a notation to make the error checking take one statement, so the programmer is more likely to do both it and *T* properly.
- **Cweb** allows the programmer to modify comments, documentation, indentation, even the entire web structure without causing the code file to be updated. This means that if **make** (a UNIX 'least effort' program configurer) is being used to construct programs, **make** will only recompile a code file after a *direct* change has been made to the code in the corresponding **cweb** file. This is, of course, very helpful when editing header files which are included in many **C** files. Without this help, even editing a comment in a header file would necessitate recompilation of all code files depending on that header, or at least entail some administrative burden.
- Even if **C** is used conventionally and then **cweb** used subsequently, the program quality is likely to improve, not least because the **C** code will be reviewed as documentation is added. Several programs which have been modified for **cweb** have had bugs uncovered *and fixed* as descriptive abstractions were introduced. No other known documentation method could have such a profound effect. Usually documenting a program is so tedious anyway that either the documentation ends up being full of half-truths or, if true, half full of bug warnings.
- To produce **C** and **troff** files, **cweb** takes about one-sixth of the time of their respective processors, **cc** and **troff**. This is negligible. Future versions could easily improve on this figure.
- The programmer can code in sensible chunks (e.g. a dozen or so lines at a time) without worrying about the run-time overheads of the chunking mechanism. Psychological considerations were some of the primary motivations for one of the direct precursors of literate programming[18].
- There are many incidental advantages too numerous to mention here; for example, commented-out **C** code is hard to mistake as executable because it is formatted differently.
- Of examples which could be given of **cweb**-in-use, perhaps the most encouraging is the improvement in the presentation of student projects which have involved considerable programming effort. 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 previously tedious internal documentation is in one place and is easier to mark.
- Finally programs (as works submitted to machines rather than to humans) are not covered by the British 1956 Copyright Act. But works of literary merit are, and it is possible that a literate program would have a much stronger claim to copyright protection than a conventional program which, in law, is not specifically intended for human appreciation.

5. THE DISADVANTAGES OF LITERATE PROGRAMMING

The most subtle disadvantage of **cweb** complements one major advantage: because it motivates programmers to document their programs, programmers will be more egoistic about their programs. This will result in all the problems which Weinberg illuminates so well[19]. We also found a reluctance to write corrections on typeset documents; a typeset document presumably looks far too good to be wrong... however, **cweb** can produce output for line printers which do not have this drawback. But the immediate disadvantages of **cweb** are not in the obvious effort in learning an extra system (which is surely compensated for by its advantages), but in learning – or, worse, by being accidentally caught-out by – arbitary conventions and the normally helpful default behaviour of **cweb**, which is intended to ease its integration with other program tools. It is unfortunate that a user of **cweb** may be affected by consequences of the curious conventions of a software tool he has no intention of using.

Cweb produces **C** source, which includes compiler directives to ensure programming errors are reported in terms of the line numbers and sections of the original **cweb** source file. (Because of the unintegrated nature of UNIX programming tools, the compilers produce diagnostics with line numbers, which the programmer must remember himself and then use explicitly with an editor to locate the offending lines.) Providing line numbers and sections is the default, but at least one source-code tool cannot handle these 'helpful' directives, so a flag is available to the **cweb** user to request less informative but more portable directives. This is a feature which need only be learned by those programmers using that particular source tool, but it is not an intuitive feature which a naïve programmer would expect to exist when he first encounters the problem, not already knowing of the facility.

Another example relates to the fact that the **C** preprocessor implements both these line-number directives and a macro-processing facility of its own. In principle **cweb** can define bodies for these preprocessor macros, but the preprocessor cannot handle the nested line-number directives inside its own macro definitions!

The arbitrariness and especially the unrelatedness of these quirks makes **cweb** much harder to learn.

Other examples could be given; they are all terribly arcane, and so far as I can see unavoidable *in principle*, at least while **C**, **troff** and the other tools as given are fixed.

Although **cweb** was described well enough in §2 to be used, the user manual runs to 40 pages (including appendices). When the details are considered, **cweb** becomes more complex (although it must be said that the greater part of the manual is concerned with typography). It is not clear that the learning-time disadvantage of **cweb** would be 'spread out' if **cweb** was language-independent, even if the same notation could be used with other languages. The difficulties in learning **cweb** arise largely through lack of precise specification of such language features as comments, directives, line continuations, significant identifier length (which is not a constant in **C**), the effects of non-printing or non-standard characters, parity, conditional compilation... Language independence is discussed further in §8.1 below.

Cweb obviously adds a CPU time overhead on program development. The current version of **cweb** takes approximately one-sixth of the time that the compiler takes to compile the generated program source to object code modules. (There is considerable scope for strategic optimisation which was not attempted in the prototype version.) No experiment has been performed to see if total program development time is reduced (because of greater programmer efficiency), which is possible, but unlikely because of the extra investment encouraged in documentation. The investment in documentation would be expected to more than pay for itself during program maintenance, which is notoriously the most expensive part of the program life cycle.

Understanding a correct program is greatly facilitated by **cweb**, and the larger the program the more noticeable the improvement, but 'small scale' on-line debugging is actually made harder (really as a consequence of the editor not being integrated into the 'web way' of doing things). For example, if a loop fails to satisfy its invariant, but the body of the loop has been abstracted away as a macro, the programmer will have to flip between two (or more) places in a file which would have been contiguous if it were not for the 'helpful' web structure. On the other hand, similar remarks can be made about any form of structuring, whether a respectable abstraction mechanism or **goto**! It behoves the programmer to name modules appropriately and to be specific about side-effects on sequencing, free variables and so on.

Because **cweb** performs no syntax analysis, it is possible to make simple slips whereby a programmer is misled by the seemingly atomic form of a macro invocation. For example, a macro may expand into a command sequence (say, 'A; B'), and execution of only the first command of which ('A') is controlled by a construct guarding the macro:

where
```
if expression then @<m@>
@<m@>= A; B
```

This problem (of referential opacity) is familiar to users of macro processors, but is still unfortunate and could be detected automatically. Indeed, the syntax analysis necessary to detect it could have been used to advantage for other features of **cweb**, such as applying scope rules to identifier fonts. (The current system binds fonts to names not identifers, (*name*, *scope*) pairs.) Incidentally, full analysis is possible only if the web is well formed, otherwise the program code is not properly specified.

6. CWEB AS A TRIVIAL INTEGRATED PROJECT SUPPORT ENVIRONMENT

Cweb combines **C** program source code and **troff** formatter instructions. It is a very simple IPSE. **Cweb** has to use a notation which permits its user reasonably unaffected access to both **C** and **troff** and, indeed, to all the other tools which may be used with these languages.

UNIX gains its success in part from the convenient integration of a variety of software tools; why then should a general-purpose operating system be more successful as an IPSE than **cweb**, which is a special-purpose system which was carefully designed to integrate only a few specific tools? The common factors in UNIX (filestore, device drivers, pipes) are as featureless and transparent as possible and (to some well-defined extent) interchangeable operands. The common factor in **cweb**, the web specification, is highly structured: it has multiple context-sensitive lexical conventions. Tools which read a web specification have to be specialised to handle this. Depending on the intended application, generating code or formatter source must retain some but not all of the quoted information: preprocessing cannot be decoupled from postprocessing. This would not be a serious problem if the various mechanisms could be hidden from the user: contrast the ease with which Pascal and FORTRAN subroutines can be linked on various systems, principally because representations (such as stacks) are not made explicit in either language.

A reasonable definition of a software tool is that it does *one* thing well and without side-effects. UNIX (at least in its early days, before people started writing big programs making use of more than sixteen bits of address space) does one thing well, namely multiplex resources. With the constraints of a small address space, program features which survive are really necessary; with address spaces which exceed our ability to fill them sensibly, 'software tools' tend to accrete gratuitous features. Software investment can be correspondingly greater (indeed it is generally impossible to tell when software development is complete, if ever) and in time it becomes quite impractical to rationalise a set of tools.

There are tools apart from **cweb** which may be applied to **C** or **troff** source, and they intrude on 'original' **C** and **troff** introducing their own conventions. Some software tools use '**@**' and '**$**' for precisely the same reasons **cweb** uses them... that 'they aren't already used'! Some tools scan comments for extra-language directives which control such features as run-time domain checking: such directives are usually idiosyncratic and rarely portable. What should **cweb** do with comments, given that not all comments are really non-functional commentary? In fact just the two tools that **cweb** primarily attempts to integrate are not even compatible.* **Cweb** has to use its

* For example, a problem arises with **C** strings since they may contain any characters (including end-of-line and quote, both **troff** argument delimiters) and there is no general method to pass such strings as arguments to **troff** functions, since (*a*) every **troff** argument is evaluated, and one cannot know *a priori* how many times a string is going to be passed as an argument in order to quote delimiters in it the right number of times for a particular context, and (*b*) there is no notation at all for embedding a newline character in a **troff** argument.

own arbitrary conventions. However, users who have already used certain tools separately or in novel combinations may have their own additional or conflicting conventions. Thus **cweb** conventions have to be programmable. Again, the result is a hacker's paradise and a very obscure collection of rules for special cases, to ensure correct treatment of the most common cases by default.

Cweb source files have to be edited. The standard UNIX editors are general-purpose, but they are less suitable for editing **cweb** source files than for other program files. **Cweb** permits name abbreviation, and no current editor permits a similar context-sensitive abbreviation for searching for occurrences of strings in a file. **Ed** is a typical editor: if we neglect the equality of **cweb** names irrespective of the disposition of blanks (which cannot be specified in **ed**, because there is no provision for matching embedded newlines), the relevant command for searching would be '/@<*name-prefix*.*@>/'. This would locate the next string of which '@<*name-prefix*' was a prefix and '@>' a suffix. In **ed** we cannot specify context constraints, such as 'not in a **C** string', where the sequence '@< ... @>' would not denote a **cweb** name. So far these difficulties are minor – they just discourage use of the more relaxed features of **cweb**. But such a search will locate the next string in the file of which '@<*name-prefix*' is a prefix, and this will miss all abbreviations in the file which are valid but shorter prefixes of the same target name. Programmers make less use of the abbreviation mechanism than they perhaps would do because of this difficulty.

Until a programming language is designed from scratch anticipating or including web-like ideas, software tools for literate programming will be doing too many things (mainly trying to be general in various incompatible contexts) to be really successful in present software development environments. A common design principle is, 'make the simple easy and the complex possible'. Sometimes it might be better to forbid the complex in the interests of making the simple easy and free of exceptions.

7. IDEAS FOR AN INTERACTIVE VERSION

Cweb is implemented as a conventional two-pass batch processor; that is, it has to examine the entire web source twice before it produces any results (it can of course produce diagnostics earlier). In an integrated interactive system this is not acceptable: **cweb** must be incremental, with time to produce results small and constant or commensurate with the last change made, not the total size of the specification. At present **cweb** provides a hierarchal notation for program code but no corresponding hierarchal constructive method for web specifications. Web specifications in the current system are linear character strings (and as such are edited using existing text editors). Although **cweb** in its current form is restricted to a kind of ASCII specification, there is no reason to retain this if high-resolution or semi-graphic displays can be used. The symbols '@<', etc. can be abstracted out by graphic conventions, such as line frames around sections. Such a system could have features tuned to the needs of **cweb** programmers – such as searching for abbreviated module names correctly, but more usefully, facilities for moving and displaying the bodies of macros at the place of their invocation. The programmer could draw a box around a fragment of code, name it 'increase *i* under invariance of P1', hit a button and the code would be 'folded away', replaced by a box indicating an invocation to the named code.

A window notation, with macro names as 'tabs' on the frames of code, can be seen to permit easily revocable display of the body of invoked macros. One of the disadvantages of literate programming already mentioned is that the programmer cannot view the code of macro bodies *in situ* at the point of invocation. In an interactive version, the programmer would press a button and an invocation would unfold into a (*name*, *body*) window in position. Note that free-text macro processing (as used by a liberal **cweb** system) would clash with structure editing, since the bodies of macros need not be syntactically well formed.

The much-worn phrase 'what you see is what you get' may be applied, but with interesting consequences. In the web case, 'what you get...' may be either of two objects – a program or a literate program document. The programmer may want to work on either view of the web: for debugging he would need an unobstructed view of code, and for writing documentation he might need to refer to higher-level structures, such as major section names and sequencing in the web. Some changes the programmer will want to make permanent, and some will be transient 'changes in perspective' which are not intended to restructure the hard-copy/documentation version of the web.

An interactive version would be much easier to make language-independent, since the user interface could constrain the user to constructing specifications in a well-defined subset of the domains of the various software tools involved. As a very trivial example, the interface could prohibit lines longer than a certain (display-related) constant. With **cweb** as it stands, **cweb** itself has no control over the production or form of **cweb** specifications and so it must be able to handle any correct **C** however it comes.

Feiner, Nagy and van Dam point out that a user might easily get lost in this sort of uniformly presented but semantically rich system without the cues over several sensory modalities that would be expected in a real-life system of similar complexity[3]. Thus a paper book can use colour, pictures, index, running titles, layout, fonts, bent-over corners, annotation and so on as cues to orientate the user. Feiner *et al.* suggest that an interactive system of the form considered here should provide the following.

Annotation: a method so that the user can add (preferably handwritten, drawn or otherwise distinctive) notes 'in the margins'.

Folio: a standard display format which includes explicit information capturing the progress of the user dialogue, such as the current time and an iconic representation of the last folio. All displays will be in folio.

Timeline: a canonical representation of the user's actions, probably shown as miniatures of folios, over a period of time, ordered left-to-right to show their sequencing.

Index: various facilities so that the user can locate folios by abstractions.

Neighbours: the ability to view adjacent folios to the current folio, either by miniaturisation of all folios or

using a 'bifocal' method such as that advocated by Spence and Apperley[14].

Colour: as an extra cue, perhaps to indicate the permanence of changes in perspective or to facilitate cross-referencing index entries.

An interactive literate programming system would also borrow ideas as exhibited in such systems as Mentor (Donzeau-Gouge, Kahn, Lang & Mélèse[2]) and XS-2 (Sugaya, Stelovsky, Nievergelt & Biagioni[15]). Work is in progress at the University of York to define one, and our particular interest is the behavioural evaluation and formal definition of a consistent system *from first principles*[4].

7.1. 'Literate using'?

Literate programming combines the normally separate activities of programming and documenting. As we have seen above, this activity itself may be made interactive, and indeed it is desirable to do so. But the literate programming approach is much more general and could be applied even when the underlying programming language is interactive in its own right. By analogy with the term 'literate programming', combining documentation with using could be called 'literate using'. A literate using system would document running programs. As it stands at present, **cweb** only helps construct and maintain the *internal* documentation of programs, and the external documentation of a program's use is likely to be as flawed as ever. There is no reason why similar notions could not be developed to help write user manuals or other forms of system exposition.

There are several combinations of possibilities. For example, the system being documented is fixed (we are then writing a conventional user manual) or the system is under development (for example, the programmer is defining a menu hierarchy). Then the 'manual' is either conventional (intended to be printed on paper) or is itself interactive, possibly even as an integral part of the system being documented (e.g. 'on-line guidance'). With some imagination, such 'literate using' ideas may be developed in order to provide live demonstrations of interactive systems – perhaps under user direction. It is entertaining to construct a matrix of the combinations. For example, it would be reasonable to have a system which enabled a programmer to write a book about programs; he will want to document the code fragments (literate programming) and show example output by running the actual code included in the book (literate using). Indeed, right now you are reading a paper about a literate programming system which includes example input and output generated by that system.

User manuals typically include examples of the form, 'If *i* is input when the system is in such-and-such a state, *o* will be output'. It is very tedious keeping such examples up to date. It should be a simple matter to include suitable checking facilities in a literate using system and facilitate the task of updating the manual where necessary. Equally it would be useful to confirm automatically whether a manual remains correct despite modifications to the program it documents.

A number of interesting issues have been raised from preliminary work with my colleague Colin Runciman. For example, one is immediately faced with a design choice: should a literate using system facilitate the construction of *any* expository text, or should it impose a formal structure on the exposition? We tend to the latter view. The system constructs expositions in which formal and informal text are clearly distinguished; thus the reader can be certain that formal text corresponds *exactly* to the behaviour of the documented system. This is stronger than the current literate programming approaches but has other advantages: for example, it permits a program reading the *same* document to verify the formal parts, perhaps against a new version of the documented program. No doubt such strictness would lead to a stylised form of user guide, but it would be premature to dismiss the approach for a superficial lack of generality.

UNIX passes Doug McIlroy's operating system test for file system uniformity (Ref. 5, page 47); perhaps it is time to add the 'literate using' test – for, in keeping with the experience reported elsewhere in this paper, we have encountered non-trivial problems implementing the idea under UNIX. In fact, a practical literate using software tool cannot be implemented under UNIX without modifying the operating system kernel – there are a number of process-synchronisation problems.

8. EXTENSIONS WITHIN THE CURRENT FRAMEWORK

Cweb is less intrusive in C than WEB is into Pascal, but there remains a range of ideas which have yet to be explored. On reflection it seems that those features provided by **cweb** are a minimum for its purposes, but any more would make **cweb** appear considerably more daunting. C and Pascal are similar languages in many respects; literate programming is likely to evolve along different lines for languages such as Prolog (which has very little structure but is more sensitive to sequence) and Ada (which has much more structure). Languages which permit the definition of operators would probably need auxiliary definitions for operator formats.

The keywords of **cweb** have been chosen to be brief and orthogonal to the base language. Obviously, if I had wanted to intrude more into C, the notation could have been much more pleasant and less liable to conflict with that of other software tools. If macro bodies were restricted to command sequences (or, better either simple or compound commands), not only could one of the problems discussed above (§5) be avoided but the invocation syntax could be simplified. For example, the operators '<' and '>' are always binary (on expressions) in C, so the lead-in '@' could be dropped without introducing ambiguities. (C provides its own macro system which is more suitable for expression macros, so this restriction would not be at all serious.)

It is worth noting that C (unlike Pascal) uses semicolons as command terminators; this allows all **cweb** macro bodies to be terminated by a semicolon (or to be compound commands) irrespective of the context of their calls.* In Pascal, if a command is followed by **else**, it cannot terminate with a semicolon. This is a non-intuitive (but formally as consistent) since in all other cases in Pascal a semicolon is either required or is

* However, a C preprocessor macro is not at such liberty: if '`f(x)`' is a macro or function invoked in the context '`if( x ) f(x); else ...`' then '`f`' cannot be defined as a macro with a compound command body.

optional. This feature of Pascal somewhat restrains restructuring programs.

It is possible to provide a mechanism (which can be simulated in the current **cweb**, but is not standard) to specify the formatting of vertical spanning units: if a section is too big to fit on a page, the programmer may want to indicate where the section may be split conveniently (with varying penalties) to avoid 'logical' widows. One could argue that if the spirit of literate programming is seriously adopted by a programmer then such long sections would never arise in practice, but adherence to the first design principle – that previously valid programs should still be valid (irrespective of the length of their components) – suggests that an arbitrary maximum size would be too restrictive, however generous. Also, of course, the section size includes the documentary parts and these may be substantial on their own, posing the usual typographical problems of widows and orphans at page boundaries. As anyone knows who has taken typography seriously, there are many typographical controls which could be provided, and these are by no means unrealistically complex if high-quality presentation is required. However, there is a valid school of thought (exemplified in the philosophy of SCRIBE[12]) that author/programmers need not bother or waste their time with typography. Professional literate programming systems must provide an 'author' mode without access to typographical niceties, and without the learning effort (and temptation).

A more serious problem is how to handle closely related sections. For example, **C** and most programming languages require definition-before-use, but the flexibility of the **cweb** notation soon obscures whether definitions precede or follow applications. To get around this, the programmer will define a macro to accummulate forward definitions (by using multiple definitions to append to a single code body – see §2.1). Since **cweb** is two-pass it could in principle provide the necessary topologically sorted forward definitions itself (or, better, a separate software tool could do the job). For the present version of **cweb**, however, the problem appears in the choice of formatting convention to be used for such very closely related sections. Should a section be able to contain more than one macro definition?

Note that under the present **C** language standard, as the form of identifier bindings in initialised and uninitialised definitions is different, it is not even possible to share the same text for both purposes by using a common macro body. The scheme for permitting macro bodies to be written to separate files was intended for and has mainly been used to handle forward (uninitialised) definitions: it works, but I think it is a clumsy concept. When definitions have to be manually duplicated because of such idiosyncrasies in the language the programmer may make slips and is put in a mild quandary about which definition to document! Again, a suitable software tool could perform the purely clerical operations.

Cweb combines a pair, code and documentation, from which it can extract either. In many applications it would be useful to enhance code with other attributes which should also be documented, such as topology, pre- and post-conditions, invariants, and JCL to process the code and other such attributes. The update history (version control and bug fixes) of a program could also be included. But showing all these attributes with equal prominence would produce a very cluttered literate program document, and perhaps they are more appropriately combined in a primarily interactive system where the user can dynamically view those attributes of immediate concern. It is a feature of Mentor[2] (op. cit.) that attributes (and textual language delimiters) are not normally displayed.

Cweb assumes that the resulting literate program is self-contained (although the code generated from it may be separately compiled). It is possible that a web-like method could be used in writing a book. In that case extra handles would have to be provided on indexing, cross-referencing and so on. In fact, these are really options which should have been provided by quite separate software tools. Currently **cweb** provides tagged integer sequences for numbering sections, but it is arguable that 'levels' of sections, 1.1, 1.2.4,..., should be provided. Certainly they would be necessary for webs embedded in larger documents (with an existing numbering system) such as might occur in books on algorithms. Sophisticated number schemes are awkward to handle, since **troff** has all the right features for generalised counting (even in Roman numerals, with levels and so on) but **cweb** needs to know section numbers (for forward cross-references) *before* any of the document can be formatted by **troff**. This is an example of the standard feedback problem arising with sequential processing arrangements for 'integrated' software tools.

8.1. A language-independent version

In its current form **cweb** needs a table which specifies the programming language lexemes, quoting mechanisms and comment conventions. These features can generally be expressed using regular expressions. A format must also be provided for each lexeme or class of lexemes, for example that 'if' is a reserved word and should be formatted in a bold font, or the lexeme '<=' should be formatted '≤' or '<=' depending on preference. In **C**, there are just two context-sensitive quirks due to operators being overloaded. First the operator '*' may signify either multiplication or indirection (right value): in **cweb**, two default formats are provided – programmable, of course – namely '×' and '*' respectively. Similarly, '·' denotes either a decimal point or a field selection operator. These could be considered pedantries. Secondly, there are multiple scopes for identifiers, since field names need only be unique within their defining structure; in fact, **cweb** only distinguishes between identifiers that are field names and all other identifiers, for the purposes of distinguishing field names as such in the index. **Cweb** enforces the same font for the same name regardless of how the identifier is used. Since no syntax analysis is performed we could not distinguish between, and hence could not format differently (say) *x* as a function and *x* as a field selector. In **C** it is conceivable that programmers might want to distinguish between macro invocations (that is, applications of standard **C** '#define' macros) and function invocations of the same name. These problems are not insurmountable, but they have to be addressed in the absence of an existing publication standard which could have been adopted.

Because **troff** is not formally defined, *ad hoc* solutions had to be found for processing particular features of **C** (e.g. to **C** pass strings as **troff** macro arguments – see

footnote on page 206), and language-independent specifications could be messed up by abstruse formatting requirements. Certainly, a language-independent system should not and probably could not use **troff**.

Language independence of a more limited form is often required. Under UNIX, several software tools exist which generate **C** programs, in particular compiler-compilers which operate on specifications which may include considerable fragments of **C** code (e.g. **yacc, lex**). Generally such tools introduce additional lexemes and quoting mechanisms, and in principle could be handled by straightforward extensions of **cweb**. In some cases the choice of '@' and '$' would need to be reviewed. Currently, **cweb** provides no useful support for tools of this kind, and this is definitely an omission. Since **cweb** acts on text, there is no reason why its macro mechanism should be restricted to the target language: it can also be applied to the metalanguage of the compiler-compilers. But if the lexemes of the languages overlap (and should be distinguishably formatted), non-evaluative analysis of the **cweb** source will not reveal the correct formatting conventions for the bodies of macros, since this would be determined in the context of their invocations. For example, a macro might be defined as the text 'A | B'. This is legitimate **C** ('A', 'B' identifiers), and also the legitimate right-hand side of a **yacc** production rule ('A', 'B' symbols). With the current purely lexical font selection these uses could not be distinguished. Of course it is possible to require all code to be formatted in a uniform font, but this restriction would go against one of the current design principles.

8.2. A formatter-independent version

It seems unlikely that there would be much call for a dynamically formatter-independent version of **cweb**, as it is likely that an installation will have its own preferred formatter. However, formatter independence is easier to achieve than programming language independence since formatter text (documentation and formatter instructions) is simply copied to the formatter unchanged by **cweb**. The formatter itself should provide device-independence, such as over ink-jet/laser-printer or details such as paper size. If porting **cweb** programs is envisaged across operating systems, where formatters may vary, it may be worth developing a minimal set of formatter directives, such as 'new paragraph', which can be translated to equivalent local forms by **cweb**.

In fact, the current version of **cweb** does generate a formatter-independent text stream which is translated into **troff** form by a separate co-process. The user can select alternative processes to manipulate the text stream for other purposes (although currently no very interesting one has been implemented): perhaps for producing quick-and-dirty output to a fast line printer; for producing hierarchy diagrams; for producing subsets of the document, such as just the cross-references; for using other formatters, such as TEX; and so on. **Cweb** does not attempt to pretty print **C**; if this is considered an omission it could easily be corrected by inserting a pretty-printing filter in front of the existing **troff** source co-process – although it would take two passes per module if the self-setting alignment tab scheme was retained.

9. IMPLEMENTATION PROBLEMS

Writing a program to drive another program which was originally intended to have human input was a mistake. I did not want a **cweb** user to have to rewrite his program if **troff** couldn't format it well enough! The user interface of **troff** is designed for human-written text: certainly few documents are satisfactorily produced without experimenting. If formatting a document goes awry, typically by widowing a single word, the best way to correct it is to rewrite the last paragraph more concisely. When **troff** is driven by a program, it is not possible for that program to rewrite text in order to get it to fit on a page or to satisfy any other of the many reasonable typographical constraints. In other words, **cweb** has to generate the correct **troff** commands for all input; and this turned out to be considerably harder than anticipated. About 95% of the development effort of **cweb** was consumed in **troff**-related issues. (The current system object code devoted to purely **troff** code generation, excluding checking and index sorting, totals 50k bytes as compared with 2k for generating **C** code.) It is surprising that the user interface of **troff** does not come up to even basic standards for machine–machine communication. It is amusing, perhaps, that a draft-quality formatter (as one of the standard co-processes of **cweb**) which implements *precisely* the subset of **nroff** required by **cweb** is about the same size as the code-generator co-process for **nroff** itself (and about $\frac{2}{3}$ the size of the code generator for **troff**). Telling **nroff** what to do is as difficult as doing it!

I made a strategic mistake thinking I could get away without syntax analysis, and **cweb** is somewhat harder to use because of this decision. Or maybe **cweb** is *easier* to use, since any syntax analysis would be subject to yet more hedges. The decision to have no syntax analysis was made originally to make a gain in processing speed, which was felt important for user acceptability, but in retrospect this gain is probably offset by the time wasted in occasionally submitting bad **C** code to compilers. Half a dozen '@'-keywords for typographical control are necessary because **cweb** performs no syntax analysis, but it is likely that they would be required occasionally even if **cweb** did perform sufficient analysis: they provide the programmer with useful control over indexing and fonts. Maybe if these features were used explicitly less often (because a sophisticated **cweb** system inferred and defaulted them) for the sake of 'ease of use' programmers would have a prolonged learning period before they became familiar with all of **cweb**'s functions.

One further reason for not bothering with syntax analysis was because it was hoped that there would then be no need to waste time processing included files, that is, files whose text is to be inserted at compile time into the program at various points. Since included files are used almost exclusively for defining identifiers, they are also a very natural place to define identifier fonts centrally. Thus **cweb** does have to process included files, and the original reasoning was wrong. Because the text of included files is included at compile time (by the standard **C** preprocessor) and not by **cweb**, the font specifications have to be expressed in unadorned **C**.... This notational extention can only be made in comment. Since we don't need font specifications to be explicit in the formatted literate program (which they would be if they remained as directives in **C** comment) there have to

be two mechanisms for specifying fonts. Again, the combination of tools has led to arbitrary complexity in **cweb**. Actually, the programmer rarely needs this feature of **cweb** explicitly since **cweb** automatically adds font information to all the code it generates (using the special comment notation, in fact), which it can later pick up again if a generated file is included elsewhere.

Cweb makes an attempt to reduce the size of generated **C** code files by eliminating redundant blank space. **C**, like many other languages, requires blanks to disambiguate certain constructs. It obviously requires blanks between adjacent reserved words and identifiers but there are less obvious cases. I was aware that '`x+ ++y`' (and even '`x-- - --y`') would become either ambiguous or likely to be parsed incorrectly if **cweb** removed blanks too eagerly but I have recently noticed the problems of '`x/ *y`' (intended to mean division by '`*y`', not comment-out '`y`') and '`x& &y` (intended to mean bit-and with an address, not logical-and).

10. CONCLUSION

Literate programming is promising and successful, but has a long way to go before it emerges as a mature discipline. In the meantime, literate programming is likely to be used more and more frequently in published programs, and appropriate formatting standards will evolve. The current format used by **cweb** is not sufficient for all tastes and it is also very disappointing how long it takes to specify a new style with **troff**. At some stage literate programming systems will develop as purely language-independent notations, interactive structure editors, or properly integrated into new languages.

It is surprising how badly some software is designed when its input is expected to be human-generated: the text formatter used by **cweb** is terribly inconsistent and extraordinarily difficult to drive by program. Perhaps it was never formally considered. I believe Knuth is wrong when he asserts that a WEB system is easy to implement (Knuth, 1984) – a *basic* one is easy – but aesthetic considerations of presentation are not easy to anticipate nor formalise, and it is difficult to contain the resultant system complexity.

In restrospect it seems obvious, but 'integration' cannot be retrofitted to fixed software tools without introducing hacky features. The major problem here has been the interaction and poor specification of lexical and quoting conventions in the various languages. Less severe problems arise from overloaded tools whereby a single software tool provides several unrelated features: **cweb** and other integrated systems may need more than one feature of that same tool concurrently. There is much more promise in interactive 'constructive' integrated systems, but some functionality must be sacrificed.

REFERENCES

1. R. Baecker and A. Marcus. On enhancing the interface to the source code of computer programs. *Proc. ACM Conference, CHI'83, Boston: Human Factors in Computing Systems*, pp. 251–255 (1983).
2. V. Donzeau-Gouge, G. Kahn, B. Lang and B. Mélèse, Document structure and modularity in Mentor. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 141–148. Pittsburgh, Penn. (1984). (ACM SIGPLAN Notices, 19; ACM Software Engineering Notes, 9.)
3. S. Feiner, S. Nagy and A. van Dam, An experimental system for creating and presenting interactive graphical documents, *ACM Transactions on Graphics* **1**, 59–77 (1982).
4. N. Hammond, M. D. Harrison, A. Monk, C. Runciman and H. W. Thimbleby, *Mechanisms for Specification, Implementation and Evaluation of Interactive Systems*. Alvey/SERC Grant GR/D/02317 (1984).
5. B. W. Kernighan and R. Pike, *The UNIX Programming Environment*. Prentice-Hall, London (1984).
6. B. W. Kernighan & D. M. Ritchie, *The C Programming Language*. Prentice-Hall, New Jersey (1978).
7. D. E. Knuth, *The WEB System of Structured Documentation*, Department of Computer Science, Stanford University (1982).
8. D. E. Knuth, Literate programming. *The Computer Journal* **27**, 97–111 (1984).
9. D. E. Knuth (translated by T. Kurokawa), Literate programming, *bit* **17**, 426–450 (1985). (Publisher: Kyoritsu Shuppan, Tokyo.)
10. P. A. de Marneffe and D. Ribbens, Holon programming. *International Computing Symposium*, edited A. Günther *et al.*, pp. 67–71 (1974).
11. D. C. Oppen, Prettyprinting. *ACM Transactions on Programming Languages and Systems* **2**, 465–483 (1980).
12. B. K. Reid, *Scribe: A Document Specification Language and its Compiler. Ph.D. Thesis*, Carnegie-Mellon University, Department of Computer Science (1980).
13. S. R. Smith, D. T. Barnard and I. A. Macleod, Holophrasted Displays in an Interactive Environment. *International Journal of Man–Machine Studies* **20**, 343–355 (1984).
14. R. Spence and M. Apperley, Data base navigation: an office environment for the professional. *Behaviour and Information Technology* **1**, 43–54 (1982).
15. H. Sugaya, J. Stelovsky, J. Nievergelt and E. S. Biagoni, *XS-2: An integrated interactive system*, Report KLR 84-73C, Brown, Boveri Research Centre, Baden, Switzerland.
16. R. D. Tennent, *Principles of Programming Languages*. Prentice-Hall, London (1981).
17. H. W. Thimbleby, *Literate Programming in C; Manual and Small Example*. Department of Computer Science Report, University of York, U.K. (1984).
18. E. Towster, A convention for explicit declaration of environments and top-down refinement of data. *IEEE Transactions on Software Engineering*, SE-5, 374–386 (1979).
19. G. M. Weinberg, *The Psychology of Computer Programming*. Van Nostrand Reinhold, New York (1971).

Note. Further details of the **cweb** system may be obtained from Mrs Jenny Turner, Department of Computer Science, University of York, York, YO1 5DD, U.K.