Literate Programming in Forth

Peter Knaggs

University of Paisley
High Street,
Paisley. PA1 2BE
Scotland.

pjk@paisley.ac.uk

November 4, 1995

Abstract

We look at Donald Knuth's concept of "Literate Programming," investigating exactly what it is and how it is used to assist conventional programmers. We then ask what lessons we can learn from this idea and if it is possible to apply them to Forth.

We look at the alterations needed to the system and/or Forth to allow us to take advantage of this system. Indeed do we obtain all of the advantages that Knuth claims.

Finally some examples of Forth coding techniques are given allowing us to compare the more traditional techniques with Knuth's system.

1 What is Literate Programming

Literate programming was created by Donald Knuth during the development of his TEX typesetting system. It is the combination of documentation and source together in a fashion suited for reading by human beings. In general, literate programs combine source and documentation in a single file, referred to as a WEB. Literate programming tools then parse the file to produce either readable documentation or compilable source.

All the original work revolves around a particular literate programming tool called WEB. Knuth says [Knu92]:

> The philosophy behind WEB is that an experienced system programmer, who wants to provide the best possible documentation of his or her software products, needs two things simultaneously: a language like TEX for formatting, and a language like C for programming. Neither type of language can provide the best documentation by itself; but when both are appropriately combined, we obtain a system that is much more useful than either language separately.
>
> The structure of a software program may be thought of as a web that is made up of many interconnected pieces. To docu-

> ment such a program we want to explain each individual part of the web and how it relates to its neighbours. The typographic tools provided by TEX give us an opportunity to explain the local structure of each part by making that structure visible, and the programming tools provided by languages such as C or Fortran make it possible for us to specify the algorithms formally and unambiguously. By combining the two, we can develop a style of programming that maximizes our ability to perceive the structure of a complex piece of software, and at the same time the documented programs can be mechanically translated into a working software system that matches the documentation.

Knuth's original WEB tool provided a meta-language that allowed the author to combine TEX typesetting instructions and PASCAL program instructions. Two tools where provided, one converted the WEB file into a TEX file, which could in turn be typeset via Knuth's TEX typesetting software. The PASCAL instructions where printed stylistically, allowing the reader to quickly identify keyword, variables, etc. The order of the WEB file is not effected.

The second tool extracts the PASCAL instructions from the WEB file, automatically re-arranging and expanding the chunks into the correct order necessary for the compiler. The resulting file is compiler-ready and is not structured for human consumption.

There are now a large number of WEB tools for specific languages, these include Ada, APL2, C and C++, Fortran, Lisp, Maple, Modula-2, Pascal and Scheme. These tools use a number of different typesetting engines, including TEX and LATEX, HTML, roff and unix 'man' page format. There are a few language/typesetter independent tools [Tho95], most notably Norman Ramsey's noweb and the CLiP tool by Eric W. van Ammers and M. R. Kramer. Indeed Eric van Ammers claims [vAK93]:

> I have proposed for many years that programming has nothing to do with programming languages, i.e., a good programmer makes good programs in any language (given some time to learn the syntax) and a bad programmer will never make a good program, no matter the language he uses.
>
> The idea of a literate program as a text book should be extended even further. I would like to see a literate program as an (in)formal argument of the correctness of the program.
>
> Thus a literate program should be like a textbook on mathematics. A mathematical textbook explains a theory in terms of lemma and theorems. But the proofs are never formal in the sense that they are obtained by symbol manipulation of a proof checker. Rather the proofs are by so called "informal rigour", i.e., by very precise and unambiguous sentences in a natural language.

The CADiZ tool [YSE94] takes this idea further in that it allows one to formally specify and derive an Ada program by semi-automatic means. The final document being a full description of the process. A tool is provided to extract the Ada program code into a compiler ready form.

2 How does this differ from verbose commenting

There are three distinguishing characteristics of literate programming. In order of importance, they are: flexible order of elaboration; automatic support for browsing; and typeset documentation, especially diagrams and mathematics.

2.1 Flexible order of elaboration

This enables the author to divide the source program into chunks and write the chunks in any order, independent of the order required by the compiler. In principle, you can choose the order best suited to explaining what you are doing. More subtly, this discipline encourages the author of a literate program to take the time to consider each fragment of the program in its proper sphere, e.g., not to rush past the error checking to get to the "good parts." In its time and season, each part of the program is a good part.

The reordering is especially useful for encapsulating tasks such as input validation, error checking, and printing output fit for humans — all tasks that tend to obscure "real work" when left inline.

2.2 Automatic support for browsing

Tools are available for manipulating literate programs, including the automatic generation of a table of contents, index, and cross-reference of the program. Cross-reference might be printed, so that you could consult an index to look up the definition of an identifier 'foo'. With good tools, you might get a printed mini-index on every page if you wanted. Or if you can use a hypertext technology, cross-referencing might be as simple as clicking on an identifier to reach its definition.

Indexing is typically done automatically or 'semi-automatically', the latter meaning that identifier definitions are marked by hand. Diligently done semi-automatic indexes seem to be best, because the author can mark only the identifiers he or she considers important, but automatic indexing can be almost as good and requires no work. Some tools allow a mix of the two strategies.

2.3 Typeset documentation

Whilst you may not use diagrams or mathematics often it would be very difficult to describe some systems without them. Diagrams and tables are very useful for summarising ideas etc.

Literate programming tools work with a number of different typesetting engine, normally TEX, roff or HTML but rather unusually WYSIWYG is not generally supported. This is because the data formats used in WYSIWYG products are proprietary, and they tend to be documented badly if at all. They are subject to change at the whim of the manufacturer. These conditions make it nearly impossible to write tools, especially tools that provide automatic indexing and cross-reference support.

People use TEX, roff and HTML because free implementations of these tools are widely available on a variety of platforms. TEX and HTML are well documented, and TEX and roff are stable. TEX is the most portable, it was also developed by Donald Knuth as a literate program [Knu86], thus it is not surprising that

the first literate programming tools used TEX as there typesetting engine.

3 Relating it to Forth

To see how these ideas can be of use to Forth, we look at the benefits they bring to traditional languages and see how these may be transfered to Forth. Knuth claims that literate programming increases the maintainability and quality of software by improving its factoring and readability, leading to the inevitable productivity and salary increase.

Factoring: With file based languages it is quite common to see a single function definition of 80 lines or more. This is normally because the function is required to hold the full text of the algorithm it is implementing. Whilst it is possible to break the algorithm down further the overhead associated with defining the relevant functions outweighs its usefulness.

To overcome this Knuth introduced a decomposition facility into his tool. It is this that allows one to break up the definition into its constituent parts without the need of defining new functions. Thus we are able to decompose the algorithm, discussing both it and its implementation in a more natural way, with the various parts being defined and discussed in their natural place. A tool is provided to collect the various parts together and reconstitute them into the correct order.

Readability: By allowing us to use a more natural literary style of writing to describe the application we are free to discuss the design decisions and constraints that have lead to certain intricacies in the implementation. Presenting this discussion in book form allows us to break it up into discrete sections.

We are thus able to use the decomposition capability built into the meta-language to provide multiple levels of abstraction. With the book being broken into parts or chapters. We could start with a highly abstract description of the software working in more detail as we move through the book[1].

Knuth believes that creating a program should be viewed as creating a work of art. The result will automatically be more readable as the authors intentions will be laid out in much more detail. The reader will have seen the development of the software through its description. Such descriptions should be of interest to any programmer. How often have you wanted to have a quick look at the code just to see how he did that little bit?

Maintainability: Better factoring will lead to more well thought out development. The literary style of presentation allows us to not only lay out the software better, but to discuss the algorithms and their intricacies in detail. When an alteration is required it should be fairly obvious which part, chapter or section of the book will need to be altered.

As the system has been fully described we can read the intention of the original author. We are then required to re-write the relevant section of the book to reflect the desired alteration. This technique is

[1] With the development of object-oriented programming and intelligent agents it would be possible to introduce the agents as characters in a novel, developing their character through the novel until eventually we know all of the details of the characters (agents) and their relationships.

not of much use for run of the mill debugging, but is much more useful for long term maintenance.

Quality: As we are now dealing with a document rather than a program we can apply more normal quality control techniques. Tools allow us to extract program code from the document thus we are also able to apply normal programming quality control techniques.

With better factoring and documentation it is inevitable that we will be able to understand it better. Along with this understanding comes improved maintainability. If the software is easier to maintain, is better documented and better structured, this has to lead to better quality software. This has a knock on effect in that a better quality program will command a better price and thus your salary will improve.

4 What lessons can we learn?

1. The underlying philosophy of Forth [Bro84] means that our programs are well factored. Forth's 'define before use' rubric forces us to use a bottom-up approach to our system development. By adopting the re-organising aspect of the meta-language we are able to adopt the top-down or middle-out approaches.

2. To obtain the benefits of Knuth's system he relies heavily on a change of emphasis. Moving away from program development, to the development of a maintenance manual for later use. The program just happens to appear as a side effect of developing the manual. It goes without saying that the vast majority of Forth code is very badly document, if at all. Adopting Knuth's change of emphasis when writing Forth software will bring the same benefits, primarily readability and maintainability.

It is often said that Forth, like C, is a "write only language." The adoption of literate programming has done much to make C a readable language, and as a consequence made it more maintainable [KL93]. If literate programming can transform a cryptic notation such a C, it must surely be able to make Forth more readable, and thus more maintainable.

5 Comparison

In order to compare Literate Programming with other, more traditional, methods of programming we look at a sample fragment of code using the traditional and new methods. The code chosen is a small fragment (the memory management section) from a network "chat" application [Kna95] written a few years ago. We first view this as a traditional Forth block (Appendix A), then as a source file (Appendix B) and finally in a simple example of literate programming (Appendix C).

5.1 The Forth Block

In Appendix A we see the code as a traditional Forth screen, or block. The first line is an index line, used to give a general title to the block, this has lead to the practice of collecting related definitions into the same block or sequence of blocks. In the example we have the definitions relating to the memory management of the application. This practice makes it easy to find and concentrate on specific areas of code for debugging purposes.

One of the drawbacks of using blocks is that the space is limited and one is tempted to shoehorn the code and not document it. For this reason the "shadow" block was developed to allow documentation of the code presented in the associated block (as shown in the example). Normally the two blocks are printed side by side, thus making it obvious which code is being referred to in the shadow block. This provides us with an extremely useful documentation technique, yet it is surprising how many seasoned Forth programmers do not bother with the shadow block, thus increasing the difficulty of maintenance.

5.2 The File

We now see the same example as a source file (Appendix B). In order to keep some of the advantages of source blocks it is common to split a source file into a number of pages [Pel88]. As with the source block we see the first line has a summary describing the intention of the page. We can collect related definitions into a single source page.

The use of a file does release us from the space constraints of the block. This means that we no longer need to cramp the code and we can now include comments with our code much more readily. This can be seen quite clearly with the definition of `ncbs` where a single line of code has been split into three lines, each of which has an associated comment.

Source files do however have their drawbacks. As is now able to make a definition that would not fit on a single source block, it is possible to lose the good factoring that is a hall-mark of a well written Forth program. As with other file based languages it is up to the programmer to include documentation, he may include anything from a large amount of comments to non at all. It is surprising to see how comment-free source files are. This problem is shared with other languages and was one of the seeds that gave rise to the development of literate programming.

5.3 Literate Programming

Finally, Appendix C shows the code as it would appear, as a section of a book (say **1.2 Memory Handling**), under literate programming. The book would make up a full description of the application, presenting, in a simular form, all of the application code.

The comments, or commentary as we should now call it, can be clearly distinguished from the program code. This allows us to mark out the difference between a reference to a variable or definition from a general concept in the commentary.

As with the previous systems, we are able to collect all of the code relating to a single topic into a single section. Unlike the previous systems, we can also distribute code (say initialisation code) throughout the document (via the meta-language) whichever is more useful to us.

As we are using files, we still have the drawback that a very long definition could be given. This is overcome by the philosophy of literate programming which promotes the use of good factoring of problems. The decomposition mechanism was provided explicitly to help with the factoring of complex algorithms. It should be noted that we have not used the decomposition capabilities in this extract as a well written Forth program should not need them.

6 Conclusions

We have seen how Knuth's "Literate Programming" system can increase the maintainability and quality of software by improving its factoring and readability. This is achieved by a simple change of emphasis from the production of software to the production of documentation.

Can we apply these ideas to Forth? Yes, although not by simply adopting the change of emphasis. In order to assist in this change of emphasis Knuth had to introduce a decomposition mechanism into his system (via a meta-language). As Forth is well disposed to decomposition, via factoring, it would appear that we do not need to adopt this side of the system.

There is another aspect of the meta-language that is just as important to the successes of the system, this is the ability to move chunks of code around to suite the documentation. Adding such a facility to Forth would free us from the bottom-up approach imposed by the 'define before use' nature of the dictionary. Allowing us to use the top-down or middle-out approaches to software development.

With the reordering aspect of the meta-language implemented we are now in a position to make the change of emphasis proposed by Knuth. There is no reason to suppose the advantages of using this approach would not apply to Forth. Thus it would be possible to improve the readability, maintainability and quality of Forth programs. This would naturally lead on to an increase of our pleasure in programming, and as the quality of the software has improved an increase in our salary.

It is interesting to note how one of the major claims for literate programming is that it provides imperative languages with a usable factoring mechanism. Something which is fundamental to the basic design of the Forth language.

References

[Bro84] Leo Brodie. *Thinking Forth.* Prentice Hall, 1984.

[KL93] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation (Version 3.0).* Addison-Wesley, 1993.

[Kna95] Peter Knaggs. Using IBM's NETBIOS from Forth. *Journal of Forth Application and Research*, 1995. Accepted for publication.

[Knu86] Donald E. Knuth. *TEX The Program*, volume B of *Computers and Typesetting.* Addison-Wesley, 1986.

[Knu92] Donald E. Knuth. *Literate Programming.* Center for the Study of Language and Information, Stanford University, 1992. A Collection of papers on Literate Programming.

[Pel88] Stephen Pelc. Text file syntax for screen file users. In *euroFORML'88 Conference Proceedings*, pages 171–175, MPE Ltd, 133 Hill Lane, Southampton SO1 5AF, UK., September 1988.

[Sew89] Wayne Sewell. *Weaving a Program: Literate Programming in WEB.* Van Nostrand Reinhold, 1989.

[Tho95] Dave Thompson. Frequently asked questions. `comp.programming.literate` Usenet newsgroup, 1995.

[vAK93] E. W. van Ammers and M. R. Kramer. The CLiP style of literate programming. (submitted for publication), 1993. available via anonymous ftp from the CLIP directory of `sun01.info.wau.nl`.

[YSE94] Literate formal development of Ada from Z for safety critical applications. In *SAFECOMP'94 Conference Proccedings*. York Software Engineering, 1994.

A Screen

```
 1 ( NETCHAT - Reserve Memory                          PjK 10MAY90)
 2
 3 : STRING WORD C@ 1+ ALLOT ;
 4 CREATE NET-CHAT BL STRING NET-CHAT
 5
 6 CREATE ncbs   ncb_size 5 * ALLOT   ncbs ncb_size 5 * ERASE
 7 CREATE buff         60 5 * ALLOT   buff       60 5 * ERASE
 8
 9 : NCB  ( n -- a ) ncb_size * ncbs + ;
10 : BUFF ( n -- a )       60 * buff + ;
11
12 : name ( n -- a n NCB ) DUP BUFF SWAP 60 SWAP NCB ;
13
14
15
16
```

```
 1 This screen will reserve the memory buffers to be used.
 2
 3 It first defines the word STRING to compile a counted string.
 4 The word NET-CHAT is then defined as a counted string.
 5
 6 Memory is reserved for 5 NCBs (one outgoing, four incoming)
 7 Memory is also reserved for 5 buffers (of 60 chars each)
 8
 9 NCB    returns the address of NCB number n.
10 BUFF   returns the address of text buffer n.
11 A precondition for these operations is that 0 <= n < 5
12
13 The word  name  is defined to provide the parameter information
14 for a Datagram Receive on a given buffer number.
15
16
```

B File

```
\ Page 3: NETCHAT - Reserve Memory
\ Last Modified by Peter J. Knaggs 10-May-90

\ In this page we reserve the memory buffers used by the NetChat application.
\ We also define words to provide easy access to these buffers.

\ We start be defining a word (STRING) that will allow us to compile a
\ counted string literal, which we go on to use to define the common name
\ used in the application.

: STRING ( -- )
  WORD C@ 1+ ALLOT
;


\ The word NET-CHAT is then defined as a counted string.

CREATE NET-CHAT BL STRING NET-CHAT


\ Memory is reserved for 5 NCBs (one outgoing, four incoming)
\ Memory is also reserved for 5 buffers (of 60 chars each)

CREATE ncbs                 \ Define the base area
       ncb_size 5 * ALLOT   \ Reserve space for five NCBs
  ncbs ncb_size 5 * ERASE   \ Clear the memory like a good boy

CREATE buff                 \ Define the base address
         60 5 * ALLOT       \ Reserve space for five buffers
   buff  60 5 * ERASE       \ Clear the memory


\ We now define two words NCB and BUFF to return the address of the given
\ ncb or buffer.  They take one item from the stack which must be between
\ 0 and 4 (inclusive), giving access to five ncbs or buffers.

: NCB  ( n -- a )
  ncb_size * ncbs +             \ NCBs are ncb_size characters long
;

: BUFF ( n -- a )
```

```
  60 * buff +                    \ Buffers are 60 characters long
;


\ Finally (for this page) we define a new word  name  to provide the
\ parameter information for a Datagram Receive request on a given buffer.

: name ( n -- a n NCB )
  DUP BUFF SWAP 60 SWAP NCB
;
```

C Literate Program

The first part of the application is to reserve the memory buffers that are going to be used. This section not only reserves the memory but also defines words that allow easy access to this memory.

We are going to require five NCBs and buffers. We first reserve the space for the five NCBs (one outgoing, four incoming). The number of bytes to reserve is calculated by multiplying the number of bytes required for an NCB (`ncb_size`) by five. We then initialise this memory to zeros using the `ERASE` word.

```
CREATE ncbs                  \ Define the base area
       ncb_size 5 * ALLOT    \ Reserve space for five NCBs
  ncbs ncb_size 5 * ERASE    \ Clear the memory like a good boy
```

Thus the word `ncbs` will return the start address of a block of memory large enough to hold five NCBs. We now define a word `NCB` that take an NCB number (n) and returns the address (a) of the indicated NCB from our table.

```
: NCB  ( n – a )
  ncb_size * ncbs +          \ NCBs are ncb_size characters long
;
```

Now we do the same for the data buffers. This time the buffers are 60 bytes long and is given the name `buff`, while the accessing word is called `BUFF`.

```
CREATE buff                  \ Define the base address
         60 5 * ALLOT        \ Reserve space for five buffers
   buff  60 5 * ERASE        \ Clear the memory

: BUFF ( n – a )
  60 * buff +                \ Buffers are 60 characters long
;
```

We now define the word `name` that takes an NCB number (n and initialises the stack ready for a NetBios call to the *Datagram Receive* function, placing the corresponding buffer address (*buff*), the maximum size of the buffer (*60*) and the indicated NCB (*NCB*) on the stack.

```
: name ( n – buff 60 NCB )
  DUP BUFF SWAP NCB 60 SWAP
;
```