A FRAMEWORK FOR LITERATE PROGRAMMING

Federico de Ceballos
Universidad de Cantabria
federico.ceballos@unican.es

September, 2002

Abstract

This paper presents an approach that can be used to organise and structure Forth source code together with different kinds of annotations. Due to its flexibility, Forth is ready to use most of the advantages advocated by the 'Literate Programming School' without need for changes in the language or sophisticated tools.

This approach uses a tree structure for the text, dividing each node into up to five sections (Specification, Prologue, Epilogue, Validation and Rationale).

In order to test the method, the author has developed an editor using this schema that allows a Forth system to import the code.

Literate Programming

The term "Literate Programming" was coined by Donald E. Knuth in the early 80's to refer to a new programming methodology that allowed better documentation of programs than what could be found at that time. Knuth intended that "instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate in explaining to *human beings* what we want a computer to do" [1].

Furthermore, "the practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style. Such an author, with thesaurus in hand, chooses the names of words carefully and explains what each word does. He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding, using a mixture of formal and informal methods that reinforce each other"[1].

Knuth proposed a system called WEB, in which a common file was used for both documentation and code. One program (WEAVE) allowed to prepared a TEX file that could be fed into his documentation system. Another program (TANGLE) was used to extract the source code so that it could be read by the Pascal compiler.

The WEB system also included a preprocessor, something useful in Pascal but less powerful than the facilities already available in Forth.

One of the main advantages of this approach is that the code can be split into different sections and each section can be comprehended on its own. This is not needed in Forth,

1 This paragraph has been slightly edited, since the concept of a word is not available in Pascal (Knuth's first chosen language for coding). He was referring only to variables.

a language that allows and encourages definitions so short that no further splitting is necessary. The idea of using *modules* or *sections* as logical subdivisions is commonly used in Forth.

In conclusion, we find an original file more difficult to understand than when not using this method, a documentation of quality and a source code not intended for human consumption.

Elucidative Programming

Not every programmer is so keen as Knuth in written documentation. Lots of programmers prefer to browse through the code directly in the computer, especially when provided with the right tools.

Elucidative programming [2] is based on the following requirements:

> *The internal documentation must be oriented towards current and future developers of the program.*
>
> *The internal documentation must address explanations that maintain the program understanding and clarify the thoughts behind the program.*
>
> *The program source file must be intact, without embedded or surrounding documentation.*
>
> *The programmer must experience support of the program explanation task in the program editing tool.*
>
> *The program "chunking structure" follows the main abstractions supported by the programming language.*
>
> *The documented program must be available in an attractive on-line representation suitable for exposition in an Internet browser.*

In his paper, Kurt Nørmark suggest a framed layout for the elucidator with three parts: menu and index, documentation and program. The whole text is transformed into a set of HTML pages by means of an automatic tool.

Documenting code in Forth

The classical approach

Even in its first incarnations, Forth has given the programmer ample opportunities to document code.

In the chapter about *Elements of Forth Style* in [3], Brodie describes a good application as hierarchical. This sort of application may consist of:

> **Screens**: the smallest unit of Forth source
>
> **Lexicons**: a few screens that together implement a component.
>
> **Chapters**: a series of related lexicons.
>
> **Load screens**: a table of contents for the chapters.

The list should be completed with **words** (a smaller amount of code than a screen).

It should be noted was the term **screen** is based in a hardware related concept. All other terms are based on logical concepts.

Shadow screens

If shadow screens are used, the user is provided with pairs of memory blocks for both code and related information. With the aid of an appropriate command in the editor, it is quite easy to move quickly between one an the other.

This approach has the disadvantage that both parts have the same size. For a given amount of source code, we get enough space to include the glossary entries or even short descriptions, but not much more.

Other approaches

Some tools, like DOCGEN [4], can be used in order to generate manuals from Forth source code. The programmer is required to follow certain convention when coding and, as a result, the embedded documentation can be extracted without further effort. The structure of the source files, however, doesn't improve from the linear one, so that something else (a hierarchy of files, perhaps) has to be used if other sort of relationship is desired.

Description of the chosen approach

In order to use this method, the author has been developing an editor that enables the programmer to write code both structured into a hierarchy and divided into different types of data. This section gives an outline of the approach used.

A tree-like hierarchy

The full source code is divided into a series of nodes and these are put into a hierarchy. That allows the user to navigate easily and to see the information in the proper context.

The following figure shows a sample structure of a project, with two programs and a set of utilities that are shared by both programs. More descriptive names should be used for a real application.

The nodes shown in italics are *clones*, marks that refer to other nodes. When a clone is accessed from the compiler, the data found in the original node is given. When the data of a clone is modified in the editor, the original data changes.

```
Utilities
   Utilities #1
   Utilities #2
First program
   Utilities #1
   Utilities #2
   Code 1
   Code 2
   Code 3
Second program
   Utilities #1
   Code 1
```

Different places for different pieces of information

The data associated to each node of the hierarchy is further divided into up to five different sections. These are:

Specification
: This section contains the glossary entries and additional definitions associated with the code that appears in the next two sections. It is ignored by the compiler.

Prologue
: This section contains the part of the code that must be processed *before* that of the children nodes. If there are no children, all code goes to this part.

Epilogue
: This section contains the part of the code that must be processed *after* that of the children nodes. If there are no children, this part is left blank.

Validation
: This section contains the code that has to be run in order to test the main code found in the previous two sections. Otherwise, it is ignored by the compiler.

Rationale
: This last section is available for everything that could be or interest but has not been included so far. In particular, it is the proper place to discuss the reasons behind what appears in the other sections and the possible limitations of the code.

Simplicity of the tool

This is influenced by the author's needs. He uses just a few commands in his day to day programming needs and is not interested in implementing more. Furthermore, the doesn't feel the need to introduce LaTeX or similar capabilities for the text.

Software reuse

Modern software development relies on software reuse, usually based on libraries. These can be of object or (as it is preferred in Forth) source code.

Here the position is somewhat different, in the idea that the code should be tailor to a particular need, independent of the most general solution available en the library. In order to achieve this, the information from the library or the available literature should be copied into the file and adapted.

Multiple programs in a project

With the advantages of the tree in hiding unwanted information, several independent programs can be put together in a file, possibly sharing some components.

In a way, this amounts to software reuse, but only among modules in the same project.

Accessing the data

In order to process the data, an external program (the Forth Compiler) has to open an internal socket to the editor and send a command to fetch data in one of the modes: **normal code**, **validation code** or the **full validation suite**.

During normal usage, to process a node the Prologue is sent first, then the children of the node are recursively processed in the same order as they appear and finally the Epilogue is sent. To include the code of the second program in the example, the order would be:

Second program (prologue)
Utilities #1 (prologue)
Utilities #1 (epilogue)
Code 1 (prologue)
Code 2 (prologue)
Code 2 (epilogue)
Code 1 (epilogue)
Second program (epilogue)

In order to validate a node, the Prologue is sent first, then the children of the node are recursively sent as in the normal usage case, then the Epilogue is sent followed by the Validation. In this case, the order of loading for the same example would be:

Second program (prologue)
Utilities #1 (prologue)
Utilities #1 (epilogue)
Code 1 (prologue)
Code 2 (prologue)
Code 2 (epilogue)
Code 1 (epilogue)
Second program (epilogue)
Second program (validation)

During the validation stage, the full validation suite can be run. As this case the order is as follows: First the Prologue is sent first, then the children of the node are recursively validated in the same order as they appear and finally the Epilogue and Validation are sent. The order for our example would be:

Second program (prologue)
Utilities #1 (prologue)
Utilities #1 (epilogue)
Utilities #1 (validation)
Code 1 (prologue)
Code 2 (prologue)
Code 2 (epilogue)
Code 2 (validation)
Code 1 (epilogue)
Code 1 (validation)
Second program (epilogue)
Second program (validation)

Ideas for the future

The editor is being used at the moment and . Further enhancements are possible and will be made as time permits. Some of them are:

The external compiler should be able to position the cursor at a particular location

of the file, so that errors can be corrected more easily. Some other command could also be implemented so that the text can be manipulated from the Forth environment.

The whole editor can be integrated into a Forth compiler. In this case, only one program would be needed.

Rich text can be introduced into the editor. It will be shown to the user and printed, but will be stripped when processing the data.

Other types of node can be used. In particular, a node could contain an image in one of the most common formats. This would allow some resources to be put into the executable without need of additional files.

An example

Let us suppose that we are interested in solving the classical problem of converting dates between the usual day-month-year form to the number of days elapsed from a certain date.

This problem can appear to be a generic one, so that a good solution could be saved into our bag of tricks for further usage in the future. As it is, there are several questions to be answered before we should use such a generic code: Are we measuring time in days or in seconds? Which is our reference for counting? What range shall we need? Can we assume a 32 bit implementation?

Because of this, we could tailor our solution as a simplification of a more general case.

Specification

This part contains the glossary of the words that appear in the code parts. It may contain further informations about types or definitions.

```
A 'date' is the number of days elapsed since 31/12/1899 (it was a
Sunday). The valid range goes from 01/03/1900 to 28/02/2100.

DMA> ( day month year -- date )
   Transforms day-month-year into a date.
>DMA ( date -- day month year )
   Transforms a date into day-month-year.
```

Prologue

This part contains the normal code.

```
: DMA> ( day month year -- date )
   4 + OVER 3 < 1 AND - 1461 * 4 /    ( D M c )
   SWAP 9 + 12 MOD 153 * 2 + 5 /      ( D c d )
   + + 695377 - ;
```

```
: >DMA ( date -- day month year )
   695376 + 4 * 3 + 1461 /MOD         ( T'*4 Y' )
   SWAP 4 / 5 * 2 + 153 /MOD          ( Y' 5*D' M' )
   SWAP 5 / 1+ SWAP                   ( Y' D M' )
   2 + 12 MOD 1+ ROT                  ( D M Y' )
   4 - OVER 3 < 1 AND + ;
```

Epilogue

No epilogue is needed in this case, as the node doesn't have any child attached.

Validation

In this part, we can put whatever code we find necessary for the testing.

```
\  SHOW formats the 3 components of a date
\  ZZZ shows both ways in the conversion for a date

: SHOW ( day month year )   ROT  3 .R  SWAP  3 .R  4 .R ;
: ZZZ ( date )   DMA> DUP 10 .R  >DMA SHOW ;

CR .( 01/03/1900 ->    60 )  1 3 1900 ZZZ
CR .( 01/01/2000 -> 36525 )  1 1 2000 ZZZ
CR .( 28/02/2100 -> 73108 ) 28 2 2100 ZZZ
```

Rationale

One of the reasons for choosing this particular example has been that the code is difficult to write in a self-documenting fashion. The user will require additional comments, that will be different if he or she is learning to use the code, validating it or changing some of the assumptions.

This is also a good place to put external references, in this case [5].

```
The code is based on the algorithms found in 'Mapping Time, The
Calendar and its History' by E.G.Richards, chapter 25. Algorithms
E (page 323) and F (page 324) are used with data from tables 25.1
(page 311) and 25.4 (page 320).

To convert a date D/M/Y into a day number J:

Y' = Y + 4716 - (12 + 3 - 1 - M) / 12
M' = (M - 3 + 12) mod 12
D' = D - 1
c  = (1461 * Y' + 0) / 4
d  = (153 * M' + 2) / 5
J  = c + d + D' - 1401

for Gregorian-type calendars, replace last step by:

g  = 3 * ((Y' + 184) / 100) / 4 - 38
J  = c + d + D' - 1401 - g

To convert a day number J into a date D/M/Y:

J' = J + 1401
```

```
Y' = (4 * J' + 3) / 1461
T' = (4 * J' + 3) mod 1461 / 4
M' = (5 * T' + 2) / 153
D' = (5 * T' + 2) mod 153 / 5
D  = D' + 1
M  = (M' + 3 - 1) mod 12 + 1
Y  = Y' - 4716 + (12 + 3 - 1 - M) / 12

for Gregorian-type calendars, replace first step by:

g  = 3 * ((4 * J + 274277) / 146097) / 4 - 38
J' = J + 1401 + g
```

Acknowledgements

The look of the editor and the commands used for navigating are partially taken from Leo, a programmer's editor described in [6]. The idea of using clones in order to access the same data from different places also comes from Leo.

References

[1] Donald E. Knuth. *Literate Programming*. The Computer Journal, 1984.

[2] Kurt Nørmark. *Requirements for an Elucidative Programming Environment*.

[3] Leo Brodie. *Thinking Forth*. Fig Leaf Press, 1984.

[4] Stephen Pelc. *The DOCGEN documentation generator*. EuroForth 2000.

[5] E. G. Richards. *Mapping Time. The Calendar and its History*. Oxford University Press, 1998.

[6] Edward K. Ream. *Leo User's Guide*. http://leo.sourceforge.net/