fmweb: Literate Programming using FrameMaker

Version 1.0
Sriram Srinivasan
(*sriram@tcs.com*)

Introduction

This document explains a simple approach for doing literate programming using FrameMaker, a word processing system. It is almost identical to the approach taken by WinWordWEB [2], except that it can additionally be used in batch mode.

All implementations of literate programming provide facilities to break a piece of code into digestible chunks which can be documented individually, and explained in any order. The only difference is in the way the document should be written so that a tool can separate the code from the rest of the documentation. WEB and such other tools provide a way of doing “in-band signalling”, to borrow a term from communications. The programmer annotates different parts of a document by keywords, and a program such as weave or tangle recognizes these keywords. In addition, these programs provide features such as section numbering, indexing etc. and language specific features such as pretty-printing, macro expansion etc.

The other approach is to do “out-of-band signalling”, where the formatting information is essentially kept out of the documentation. This is the approach taken by fmweb and WinWordWEB.

There are a couple of reasons why I don’t use WEB based tools.

- It is always nice to see the output of a literate programming effort, but the input is rather messy. I dislike seeing a lot of meta characters (like @, <<, \, [[ ) etc. The sections are not immediately visible - one goes around looking for “\section”, for example. I do a fair amount of coding in perl, and the two notational structures put together make my head swim, esp. when I need to alter the code a few months later. Having access to a WYSIWIG word processor such as FrameMaker means that I don’t use TeX, and it is too much of an effort to learn the intricacies of LaTex or TeX to be able to use the WEB based tools.
- A number of these tools are language specific. noweb is one exception to this rule, and I rather like the simplicity of noweb. I don’t use noweb because of the reason explained above.

FrameMaker, like other good word processors, has excellent document formatting capabilities, cross referencing and indexing, table of contents, graphics, tables etc. What I see on the screen is the final output, uncluttered by meta-information.

One problem that I have with this approach is that I am used to a Unix like development environment and extensively use text based tools such as emacs/vi, perl, diff, sccs etc. For this reason, I have not tried this approach on medium or larger sized implementations, though I hope to do that in the near future.

In general, I am somewhat unclear on how the literate programming paradigm helps a developer to think. The prevailing wisdom is that this approach forces a lot more disciplined thinking on the part of the programmer, since (s)he is in a teaching mode. I find that this helps in structuring the program well, once one knows what one should be doing, but if I don’t have a top-down (step-wise refinement) solution in my mind, I am not entirely sure how to start from a blank page. I tend to do small sections of the code using my regular development environment, and once I have a reasonable top-down solution in mind, copy it all over to a FrameMaker file and start refining it.

The fmweb approach

To separate the code from the rest of the stuff, the developer uses three paragraph formats.

- “File”: The file name is mentioned in this paragraph, and all subsequent lines will eventually be dumped into this file.
- “chunk”: A chunk (or section) is a heading followed by a set of “code” paragraphs. The “chunk” paragraph tag identifies the heading for that section. It is possible to have more than one chunk with the same name.
- “code”: The developer writes code (in any language) using this paragraph tag. The Frame Maker cross reference facility has to be used to refer to chunks.

fmweb is a perl script that needs the FrameMaker document in a MIF format - it extracts chunks and code paragraphs out, and does the requisite tangling (restoring the order of chunks). All chunks with the same name are concatenated in the order they appear in the document. fmweb ignores all formatting information. Source line number information can be optionally output, in a format accepted by cc (#line).

References

1. “Literate Programming”, Donald Knuth, Center for the Study of Language and Information, Stanford University, ISBN 0-937073-80-6
2. WEB: http://info.desy.de/user/projects/LitProg/HTML.html - A pointer to all the other good stuff available on the net.

Unpacking the distribution

This document contains an example program (wc.c) written in the format explained above, and the fmweb script, along with some internal documentation.

Save this file as wc.mif. This file will be used to test out the fmweb script.

To extract fmweb (a perl5 script), go to Special->Conditional Text -> Show/Hide..., and hide the conditional tag called “Fluff”. Only the perl script should be remaining. Save this file as text only, and call it fmweb.

Run fmweb - “perl5 fmweb < wc.mif”. This should produce a file called wc.c in the local directory. Compile and execute wc, and compare against the system provided “wc”, if on a Unix system.

ToDo

- Support multiple files
- Largish example - map library
- Speed up
- Look at mml
- Accept filenames as arguments

Appendix A: ‘wc’: A word count program

This section demonstrates how one can do Literate Programming with FrameMaker. There are only three paragraph tags of interest - File (see “File:” below), “chunk” (all the numbered paragraphs) and “code”. Any cross reference format can be used - “chunkref” is the name of the format used in the example below. Apologies if the example looks like a ransom note - one does tend to get carried away by the availability of fonts, and the WYSIWIGness of it all.

The implementation and comments have been shamelessly lifted from Norman Ramsey’s noweb distribution (ftp://bellcore.com/pub/norman/www/noweb/intro.html). This is just to provide a comparison of the two approaches. I have modified some of the comments that do not apply to this approach.

Thanks, Norman.

File: wc.c

The purpose of `wc` is to count lines, words, and/or characters in a list of files. The number of lines in a file is the number of newline characters it contains. The number of characters is the file length in bytes.

A ``word'' is a maximal sequence of consecutive characters other than newline, space, or tab, containing at least one visible ASCII code.

(We assume that the standard ASCII code is in use.)

Most literate C programs share a common structure.

It’s probably a good idea to state the overall structure explicitly at the outset. Here, then, is an overview of the file wc.c that is defined by the noweb program wc.fm

1: Outline

<Header files to include> , (2:, p. 3)
<Definitions> , (3:, p. 3)
<Global variables> , (4:, p. 4)
<Functions> , (23:, p. 8)
<Main Program> , (5:, p. 4)

We must include the standard I/O definitions, since we want to send formatted output to stdout and stderr

2: Header files to include

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

3: Definitions

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

4: Global variables

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

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

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

5: Main Program

```
main(argc, argv)
      int argc;
      /* number of arguments on UNIX command line */
       char **argv;
       /* the arguments, an array of strings */
{
       <Variables local to main> , (6:, p. 4)
       prog_name = argv(0);
       <Set up option selection> , (7:, p. 4)
       <Process all the files> , (8:, p. 5)
       <Print the grand totals if there were multiple files> , (21:, p. 8)
       exit(status);
}
```

If the first argument begins with a '-', the user is choosing the desired counts and specifying the order in which they should be displayed. Each selection is given by the initial character (lines, words, or characters). For example, `-cl' would cause just the number of characters and the number of lines to be printed, in that order.

We do not process this string now; we simply remember where it is. It will be used to control the formatting at output time.

6: Variables local to main

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

7: Set up option selection

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

```
        argc--;
        argv++;
    }
    file_count = argc - 1;
```

Now we scan the remaining arguments and try to open a file, if possible. The file is processed and its statistics are given.

We use a do ... while loop because we should read from the standard input if no file name is given.

8: Process all the files

```
argc--;
do {
    <If a file is given, try to open *(++argv); continue if unsuccessful> , (11:, p. 5)
    <Initialize pointers and counters> , (15:, p. 6)
    <Scan file> , (17:, p. 6)
    <Write statistics for file> , (19:, p. 7)
    <Close file> , (12:, p. 6)
    <Update grand totals> , (20:, p. 7)
    /* even if there is only one file */
} while (--argc > 0);
```

Here's the code to open the file. A special trick allows us to handle input from stdin when no name is given.

Recall that the file descriptor to stdin is 0; that's what we use as the default initial value.

9: Variables local to main

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

10: Definitions

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

11: If a file is given, try to open *(++argv); continue if unsuccessful

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

12: Close file

```
close(fd);
```

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

13: Definitions

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

14: Variables local to main

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

15: Initialize pointers and counters

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

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

16: Global variables

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

17: Scan file

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

```
while (1) {
     <Fill buffer if it is empty; break at end of file> , (18:, p. 7)
```

```
        c = *ptr++;
        if (c > ` ` && c < 0177) {
            /* visible ASCII codes */
            if (!in_word) {
                word_count++;
                in_word = 1;
            }
            continue;
        }
        if (c == `\n') line_count++;
        else if (c != ` ` && c != `\t') continue;
        in_word = 0;
        /* c is newline, space, or tab */
    }
```

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

18: Fill buffer if it is empty; break at end of file

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

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

19: Write statistics for file

```
    wc_print(which, char_count, word_count,
            line_count);
    if (file_count)
      printf(" %s\n", *argv); /* not stdin */
    else
      printf("\n");          /* stdin */
```

20: Update grand totals

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

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

21: Print the grand totals if there were multiple files

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

Here now is the function that prints the values according to the specified options. The calling routine is supposed to supply a newline. If an invalid option character is found we inform the user about proper usage of the command. Counts are printed in 8-digit fields so that they will line up in columns.

22: Definitions

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

23: Functions

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

Incidentally, a test of this program against the system wc command on a SPARCstation showed that the ``official'' wc was slightly slower. Furthermore, although that wc gave an appropriate error message for the options `abc', it made no complaints about the options `abc'!

Dare we suggest that the system routine might have been better if its programmer had used a more literate approach?

Appendix B: fmweb source

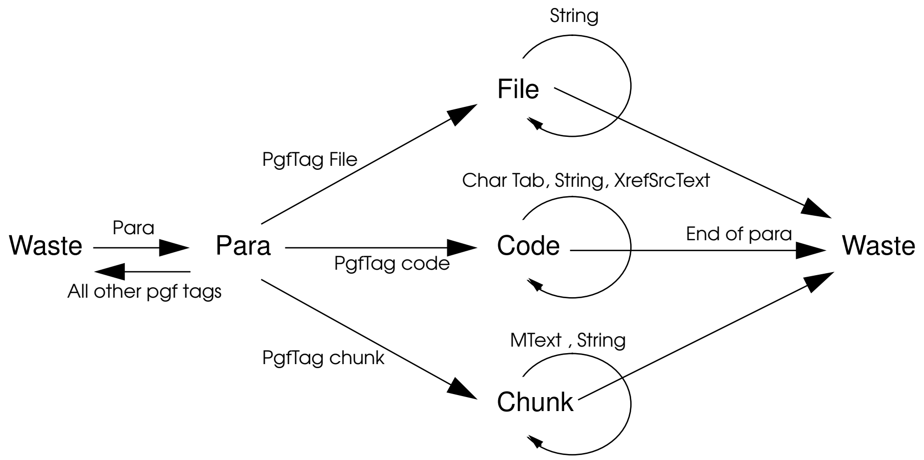
To extract this source, go to special -> conditional text, and hide the conditional tag named "Fluff". Save the file as a text file called fmweb. The first line of this file may need changing.

Table 1: Important variables in fmweb

name	Description
%chunksByName	An array of chunks, whose key is the heading of the chunk, and it's value is the entire code in that section.
$currChunkHeading	The current chunk heading
@chunkHeadings	A list of all the chunk headings, in order of appearance. If a heading has been encountered before, it is not appended to this array.
%markerReferred	Keeps track of all those chunk headings that have been referred to (by a Frame cross reference).
%markerToHeading	A translation table - Each chunk heading has a text (the "heading") that is visible, and a marker text that Frame inserts. Unfortunately, these two are not necessarily identical. This translation is necessary since the script allows more than one chunk to have the same name (which is then concatenated), but a cross reference can be created to only of these paragraphs.
$line	Global var that contains the current line from the input MIF file.
$currMarkerText	While evaluating a chunk paragraph, this keeps track of the current marker text. Subsequently, the text of the paragraph and this marker text are related by %markerToHeading
$currModeProc	A reference to a mode subroutine. Each of these subroutines is part of a state diagram (shown below).
@lookFor	The set of tokens in which this state is interested.
%chunkReferred	This array (keyed by chunk heading) tracks all chunks that have been referred to, by other chunks. The chunks that don't belong to this category are the top-level chunks.

Description

fmweb is implemented as a simple state machine, whose state diagram is shown below.



Each of the events in the state diagram are mif statements. The subroutines ParaMode, FileMode, CodeMode, ChunkMode and WasteMode implement the states. If these subs are called without any parameters, they do the necessary initializations required for that state, such as initializing @lookFor, which is the set of lookahead patterns in which the current state is interested. The main loop calls these subs with $line as the parameter - $line contains some event deemed interesting to the current state.

The script first reads in all the chunks and stores them in chunksByName, keyed by the chunk heading. Each chunk may contain embedded cross references to other chunks. In MIF, an XrefSrcText statement is the pointer to the source paragraph - more precisely, the chunk heading that this XrefSrcText points to, will have a corresponding “MText” statement with exactly the same string. Since a chunk is simply all the “code” paragraph strings concatenated together, cross references are simply embedded into this stream, with a "\200" character on either end, to identify it later.

Finally, each of the top level chunks are taken in order (top level chunks are those that haven’t been referred to by any other chunk), and expanded. The expansion process consists of substituting each embedded reference by the appropriate chunk - this is a recursive process, since the chunk referred to may itself contain other embedded references.

```
#/usr/local/bin/perl5

#forward declarations.
sub ParaMode;
sub FileMode;
sub CodeMode;
sub ChunkMode;
sub WasteMode;

Initialize();

# Start with the Waste State.
WasteMode();
# Main loop
while ($line = <STDIN>) {
    if (InterestingLine()) {
```

```
            # Call the current mode proc (one of ParaMode,FileMode, CodeMode, ChunkMode;
            # WasteMode, or WasteOneStringMode)
            &{$currModeProc}($line);
        }
}

#All chunks have been slurped in.

# Find out which chunks have been referred to by othe r chunks.
foreach $k (keys %markerReferred) {
      $chunkReferred {$markerToHeading{$k}} = 1;
}

# Can currently open only one file.
open (F, "> $currFile") || die "Could not open $currFile for writing";

# The "tangle" process. Expand each top level chunk
foreach $c (@chunkHeadings) {
      if (! $chunkReferred{$c}) {
            ChunkExpand ($c);
            $chunksByName{$c} =~ s/\\xd((2345)) /$xlat($1)/g; #Translate smart quotes
            $chunksByName{$c} =~ s/\\\\/\\/g; #Convert \\ to \
            $chunksByName{$c} =~ s/\\>/>/g; # convert \> to >
            print F $chunksByName{$c}, "\n";
      }
}

close (F);
exit (0);

#------------------------------------------------------------------------------------------------------------

sub ChunkExpand {
       local ($heading) = @_;

       print "ChunkExpand : $heading \n" if $debugging;

       if (defined ($chunksByName{$heading})) {
            # For each embedded marker found, call CHunkExpand recursively.
            # Note the /e option below.
            1 while
             ($chunksByName{$heading} =~
                  s/\200((^\200)*)\200/ChunkExpand($markerToHeading{$1})/e);
            return $chunksByName{$heading};
       } else {
            return "??$heading??"; # To indicate an unresolved cross reference.
       }
}

sub WasteMode {
       local ($ev) = @_;
       if (! defined $ev) {
            @lookFor = ($PARA);
            $currModeProc = \&WasteMode;
```

```
        return;
    }

    ParaMode();
}

sub ParaMode {
    local ($ev) = @_;
    if (! defined $ev) {
        @lookFor = ("^ *<PgfTag `code'", "^ *<PgfTag `chunk'", "^ *<PgfTag `File'");
        $currModeProc = \&ParaMode;
        return;
    }
    print "ParaMode : $ev \n" if $debugging;
    if ($ev =~ /code/ ) {
        CodeMode();
    } elsif ($ev =~ /chunk/ ) {
        ChunkMode();
    } elsif ($ev =~ /File/ ) {
        FileMode();
    } else {
        WasteMode();
    }
}

sub FileMode {
    local ($ev) = @_;
    if (! defined $ev) {
        @lookFor = ("^ *<String", $ENDOFPARA);
        $currModeProc = \&FileMode;
        $currFile = "";
        return;
    }
    print "FileMode : $ev \n" if $debugging;
    if ($ev =~ /^ *<String.*\`(.+)\'>$/ ) {
        $currFile = $1;
    } else { # End of para
        WasteMode();
    }
}

sub CodeMode {
    local ($ev, $h) = @_;
    if (! defined $ev) {
        @lookFor = ("^ *<Char Tab", "^ *<String", "^ *<XRefSrcText", $ENDOFPARA);
        $currModeProc = \&CodeMode;
    return;
    }
    print "CodeMode : $ev \n" if $debugging;
    if ($ev =~ /^ *<Char Tab/) {
        $chunksByName{$currChunkHeading} .= "\t";
    } elsif ($ev  =~ /String \`(.*)\'>$/) {
        $chunksByName{$currChunkHeading} .= "$1"; # Concatenate code to current chunk
    } elsif  (($h) = ($ev =~ /^ *<XRefSrcText (.*)>$/)) {
```

```
        $markerReferred{$h} = 1; # Mark this chunk reference as "referred to"
        # Put the text in as an embedded reference
        $chunksByName{$currChunkHeading} .= "\200$h\200\n";
        # The next string is the string of the cross reference itself - having embedded the
        # cross reference itself, we are not interested in this string.
        WasteOneStringMode();
    } elsif ($ev =~ /$ENDOFPARA/o) {
        # Assuming that each line of the code is in one "code" para . Time to append a
        # newline
        $chunksByName{$currChunkHeading} .= "\n";
        WasteMode();
    }
}

sub WasteOneStringMode {
    # Called from CodeMode above.
    local ($ev) = @_;
    if (! defined $ev) {
        @lookFor = ("^ *<String");
        $currModeProc = \&WasteOneStringMode;
        return;
    }
    CodeMode();
}

sub ChunkMode {
    local ($ev) = @_;
    if (! defined $ev) {
        @lookFor = ("^ *<MText", "^ *<String", $ENDOFPARA);
        $currChunkHeading = "";
        $currMarkerText = "";
        $currModeProc = \&ChunkMode;
        return;
    }
    if ($ev =~ /^ *<MText (.*)>$/)  {
        $currMarkerText = $1;
    } elsif ($ev =~ /^ *<String.*\`(.+)\'>$/ ) {
        $currChunkHeading .= $1;
    } else {
        # End of chunk para.
        print "ChunkMode : $1 \n" if $debugging;
        if (! defined $chunksByName{$currChunkHeading}) {
            # Init the current chunk with blank code.
            $chunksByName{$currChunkHeading} = "";
        }
        if (! defined $headingSeenBefore{$currChunkHeading}) {
            # New chunk heading name seen.
             push (@chunkHeadings, $currChunkHeading);
             $headingSeenBefore{$currChunkHeading} = 1;
        }
        if ($currMarkerText ne "") {
            # cross reference between marker text and the actual heading
            $markerToHeading{$currMarkerText} = $currChunkHeading;
        }
```

```
        WasteMode();
    }
}

sub InterestingLine {
    # Does the line belong to any of the lookahead tokens?
    foreach $l (@lookFor) {
        if ($line =~ /$l/) {
            return $l;
        }
    }
    return undef;
}

sub Initialize {
    $debugging = 0; # Change to 1 to get a trace.
    $currFile = "-";# Output to stdout
    $PARA = '^ *<Para';
    $ENDOFPARA = '^ *> # end of Para';

    #Translation table for smart quotes.
    $xlat(2) = "\"";
    $xlat(3) = "\"";
    $xlat(4) = "\'";
    $xlat(5) = "\'";

    # Start off with a default chunk heading, in case the programmer starts off code without
    # a chunk heading
    $currChunkHeading = "        DEFAULT        ";
    $chunksByName {$currChunkHeading} = "";
    push (@chunkHeadings, $currChunkHeading);
}
```