literate programming

Moderated by Christopher J. Van Wyk

A File Difference Program

Moderator's Introduction

Proponents of literate programming ascribe many characteristics to a literate program: it is meant to be read by people; it is presented in a lucid fashion and in an order dictated by intellectual logic rather than compiler restrictions; it includes a summary of the problem and the solution, an evaluation of alternative solutions, and suggestions for modification.

Readers may well wonder whether it is necessary to have a specialized system (like WEB) to do literate programming. WEB lets one typeset the solution elegantly, provides a way to define macros with very long names, and does the bookkeeping to provide a table of cross references and an index; are these facilities essential to the enterprise?

Donald Lindsay, this month's solver, clearly believes that one can write a literate program using only standard programming technology. Each function in his solution has an informally standardized header that explains its pre- and postconditions. Lindsay's comments explain other properties that he finds desirable in a program.

Joining the debate, Harold Thimbleby, this month's reviewer, explains how using a specialized system changes the process of programming. He sees the advantages that such a system provides as essential to writing a literate program.

1. INTRODUCTION

This column describes a program which reads two text files and prints out a description of the differences. The program presented here is a simplified version of a preexisting program, which has been shortened for publication by removing all code which supported options, or which improved the program's speed or its memory needs.

The program is written in the C language, and is internally documented in the concise and precise manner which is appropriate to real programs. Although some writers find this form too terse and stylized for the purposes of presentation, I believe they do a disservice. A textbook, and a useful program, simply have different purposes. For example, a program which is "explained" at considerable length, may in fact be poorly documented—from the viewpoint of a person wishing to know some quite reasonable postcondition of a certain procedure. This column should not be taken as a "literate program," in Knuth's restricted sense. This is a column about the program, but the program itself is suitable for posting (without explanation) on *Usenet*. If space allowed, the program would have been presented in its entirety (rather than as fragments), along with a machine-generated index.

The lack of innovation in this column should not be taken as an argument against progress. Instead, it is hoped to be a demonstration of the manner in which a traditional program can be kept both precise and manageable. It should be noted that the reduced program is about 700 lines long, of which about 300 are comments and spaces. Although the program was adjusted for publication, it was not fluffed; this is indeed a real program, written for real use, in accordance with standards based on maintenance experience.

The specification was the first part of the program to be written. This was kept up to date as the program evolved, and is as illustrated in Figure 1. Figure 1 contains the program's identification, a standard copyright notice, and a functional specification. In most cases, the front of a real program contains other material. For example, well-maintained programs contain a history log, showing the dates of revisions, the names of the parties involved, and short explanations of the revisions. This is usually done according to some standard format (e.g., labelled columns, or an indentation style) and the major asset of any such format is the imposed uniformity. A virtue of keeping the log prominent is

```
/***********************************************************************
*
* diff          Text file difference utility.
* ----          Copyright 1987 by Donald C. Lindsay
*               Computer Science Department,  Carnegie Mellon University
*               Copyright 1982 by Symbionics
*
* USEAGE:       diff oldfile newfile
*
* This program assumes that "oldfile" and "newfile" are text files.
* The program writes to stdout a description of the changes which would
* transform "oldfile" into "newfile".
*
* The printout is in the form of commands, each followed by a block of
* text. The text is delimited by the commands, which are:
*
*    DELETE AT n
*         ..deleted lines
*
*    INSERT BEFORE n
*         ..inserted lines
*
*    n MOVED TO BEFORE n
*         ..moved lines
*
*    n CHANGED FROM
*         ..old lines
*    CHANGED TO
*         ..newer lines
*
* The line numbers all refer to the lines of the oldfile, as they are
*    numbered before any commands are applied.
* The text lines are printed as-is, without indentation or prefixing. The
*    commands are printed in upper case, with a prefix of ">>>>", so that
*    they will stand out.
* Input lines which are longer than MAXLINELEN characters will be chopped
*    into multiple lines.
* Files which contain more than MAXLINECOUNT lines cannot be processed.
*/

#define MAXLINECOUNT 20000         /* arbitrary */
#define MAXLINELEN  255            /* arbitrary */
```

FIGURE 1

the increased likelihood that maintainers will add to it. (Beginning programmers often avoid making entries, giving excuses such as the triviality of their change.) This program's log has been omitted to save space.

The format of the printout is different from that used by the *filcom* and *diff* programs which have been used for many years. The intention was to keep the command lines straightforward and readable, at the expense of other goals (such as acceptability to a specific editor). The text lines were printed as-is, since the display of a prefixed line may change tab interpretations, causing items originally separated by whitespace to become merged together. There is also the possibility that prefixing may make some text lines too long for the user's display medium. This format does have the disadvantage that the commands can become buried in the output text.

2. ALGORITHM AND DATA STRUCTURES

The actual program contains an explanation of the algorithm which it uses. However, its explanation is mostly a reference to the article "A Technique for Isolating Differences Between Files," by Paul Heckel, published in *Communications of the ACM*, *21*, 4 (Apr. 1978), p. 264. This method is based on the idea that some lines will be found only once in the oldfile, and only once in the newfile. A file-to-file map of line matches is kept, and these unique lines are found and marked as matched. Next, lines which are adjacent to matched lines are checked. It may be that these lines would have matched, but were disqualified because they were non-unique, that is, were found more than once in either of the files. The algorithm takes the adjacency as a strong enough reason to match such lines. The map that results will show each file to consist of blocks of matched lines and blocks of unmatched lines. The printout algorithm (which was not included in Heckel's article) uses the map to print the unmatched and moved lines.

From this sketch, we see a need for two basic data structures. First, there must be a structure which holds the lines of text, so that uniqueness can be determined, and so that the lines may later be retrieved for printing. Secondly, there must be a map which relates the two files.

This program implements the first data structure by having a symbol table package, which hides the details of its data structures from its user. The package is largely conventional, and returns "handles" so that the program gets a unique name for each unique line.

The program implements the map with global data declarations as illustrated in Figure 2.

3. THE MAIN PROGRAM

Given the function specifications shown in Figure 3, and the function signatures shown in Figure 4, then the code of the main procedure (less argument checking) can be as shown in Figure 5. This code was not constructed in any particular order, and was edited into six different places in the program. However, the pieces were constructed to be proofread as a whole. Only as a whole can it be determined if a piece of code meets its functional specifications, given that the things which it directly touches meet their functional specifications.

It is important to note the word "directly." There is nothing so frustrating as discovering that the proofreading of one procedure requires reading the implementation of others. This leads recursively to including a practically unbounded amount of information into the "proof" of any single property. Although tools can assist in searches, they may not be useful in finding ill-worded or poorly placed documentation, and they cannot find documentation that was never written.

Some readers may think that "proofs" about programs are quite theoretical and academic. Actually, proofs are a major tool of every efficient maintainer, who does not have time to understand everything about a program, but must be sure that he understands certain things very well. This leads to the attitude that documentation exists so that informal proofs are easy, and are likely to be correct.

This style of thinking can be learned quite naturally. When maintaining a program, ask yourself what proofs you have constructed. If the documentation assisted these, then it is worth studying. If the documentation was inadequate to the task, then study the inadequacy, and try to alleviate it. With this attitude, a maintainer may develop good style; without it, it is all too likely that he will learn the style that makes programs need maintenance.

4. READING THE FILES

Due to limited space, the bodies of *openfile* and *inputscan* will not be shown in this column. The inputscan routine is not as trivial as openfile, but is basically just a loop, storing characters into a line buffer. It must cope with end-of-file, and with overlarge lines, but it mostly exists to call the routine *storeline*—see Figure 6. This function body has unknown semantics until we specify *addsymbol*, so we must next deal with the symbol table.

5. THE SYMBOL TABLE

The symbol table package presents a procedural interface, defined solely by the entry points. They are as presented in Figure 7. Given this code, the body of *storeline* now has well-defined semantics and can be checked against its specification.

We will not show the internals of the symbol table package. The program uses a binary tree, which is searched by iterative descent. (This point will be dis-

```
struct info{                          /* This is the info kept per-file.     */
     FILE *file;                      /* File handle that is open for read.  */
     int maxline;                     /* After input done, # lines in file.  */
     char *symbol[ MAXLINECOUNT+2 ];  /* The symtab handle of each line.     */
     int other   [ MAXLINECOUNT+2 ];  /* Map of line# to line# in other file */
                                      /* ( -1 means don't-know ).            */
} oldinfo, newinfo;

int blocklen[ MAXLINECOUNT+2 ];
/* The above is the info about found blocks. It will be set to 0, except
 *  at the line#s where blocks start in the old file. At these places it
 *  will be set to the # of lines in the block. During the printout phase,
 *  this value will be reset to -1 if the block is printed as a MOVE block.
 *  (This is because the printout phase will encounter the block twice, but
 *  must only print it once. )
 *  The array declarations are to MAXLINECOUNT+2 so that we can have two
 *  extra lines (pseudolines) at line# 0 and line# MAXLINECOUNT+1 (or less).
*/
#define UNREAL (MAXLINECOUNT+2)   /* block len > any possible real block len */
```

FIGURE 2

```
* initsymtab       Must be called, once, before any calls to addsymbol.
* ----------

* openfile     Opens the filename for reading.
* --------     Returns the file handle.

* inputscan    Reads the file specified by pinfo->file.
* ---------    Places the lines of that file in the symbol table.
*              Sets pinfo->maxline to the number of lines found.
*              Expects initsymtab has been called.

* transform    Expects both files in symtab.
* ---------    Expects valid "maxline" and "symbol" in oldinfo and newinfo.
*              Analyzes the file differences and leaves its findings in
*              the global arrays oldinfo.other, newinfo.other, and blocklen.

* printout     Expects all data structures have been filled out.
* --------     Prints summary to stdout.

* NOTE: no routines return error codes. Instead, any routine may complain
*       to stderr and then exit with error to the system. This property
*       is not mentioned in the various routine headers.
```

FIGURE 3

cussed further in Section 8, entitled "Features and Performance.") The only unconventional aspect is some counting, which makes it possible for *symbolisunique* to compute its result.

One part of any design is the choosing of names. The reader will have noticed that the names above, such as symbolisunique, are each a series of simple words, concatenated together. This is the simplest possible method of constructing long, meaningful names, and it is adequate for this small program. The drawback in large programs is that the reader will eventually encounter a name which seems to defy analysis, or which he parses into the wrong phrase. The common solutions would be to change symbolisunique to symbol_is_unique or else to SymbolIsUnique.

The underscore method is sometimes disliked on aesthetic grounds, and was quite unreadable on many early display and hardcopy devices. It makes names longer, which caused problems in the days when compilers economized space by dealing with truncated identifiers. Also, names which become visible to debuggers, to linkers, and to other tools, often fall afoul of character set or length restrictions. (These problems are usually noticed when porting software.)

```
void initsymtab()

FILE *openfile( filename )
char *filename;

void inputscan( pinfo )
struct info *pinfo;

void transform()

void printout()
```

FIGURE 4

The capitalization method is sometimes disliked as being error-prone to type, or as being difficult to communicate verbally to coworkers. (These problems are most relevant when the language used is case sensitive, as the C language is.) There are also typographic issues, such as the lack of vertical space between upper case letters, and the ambiguity of some font families. (For example, if the upper case letter I (eye) resembles the lower case letter l (ell), then the name SymbolIsUnique becomes quite confusing.) Capitalization may also cause problems during porting, typically with debuggers, linkers, and the like.

In a large program, abbreviations eventually become necessary, although only a few abbreviations (such as *len*) will be universally understood. In general, they are not as well understood as the inventor supposes, and when carried to extremes, as in *SDlocDCl*, they are clearly inferior. It is common to abbreviate *pointer* to *ptr*, and to distinguish variables containing addresses by names such as *symbolptr*. In this small program, I have used the simpler convention of prefixing with the letter *p*, as in *psymbol*.

Some readers will have noticed that function *showsymbol* is poorly designed. It is less general than it might be, because it locates a string, but also prints it (and also knows where to print it). There are two reasons for choosing this merged functionality. The first is that separating out the printing would require another function, having only a trivial (single-line) body. In a small program such as this one, one extra function represents

```
     printf( ">>>> Difference of file
               argstrings[1], argstrings[2] );
     initsymtab();
     oldinfo.file = openfile( argstrings[1] );
     newinfo.file = openfile( argstrings[2] );
     /* note, we don't process until we know both files really do exist. */
     inputscan( &oldinfo );
     inputscan( &newinfo );
     transform();
     printout();
```

FIGURE 5

```
/*************************************************************************
 *
 * storeline     Places line into symbol table.
 * ---------     Expects pinfo-> maxline initted: increments.
 *               Places symbol table handle in pinfo->symbol.
 *               Expects pinfo is either &oldinfo or &newinfo.
 *               Expects linebuffer contains linelen nonnull chars.
 *               Expects linebuffer has room to write a trailing nul into.
 *               Expects initsymtab has been called.
 *
 *************************************************************************/
void storeline( linebuffer, linelen, pinfo )
char linebuffer[];
int linelen;
struct info *pinfo;
{
     int linenum = ++( pinfo-> maxline );     /* note, no line zero */
     if( linenum > MAXLINECOUNT ) {
          fprintf( stderr, "MAXLINECOUNT exceeded, must stop.07" );
          exit(1);
     }
     linebuffer[ linelen ] = ' ';            /* nul terminate */
     pinfo-> symbol[ linenum ] =
          addsymbol( linebuffer, linelen, pinfo == &oldinfo, linenum );
}
```

FIGURE 6

a cost (in size) that partly balances against the poorer modularity. The second and larger reason is that the symbol table package may wish to keep the lines in a compressed format, or may store long lines as several fragments. In this case, the interface chosen would have some extra convenience, since the function need not recreate the original string.

6. CONSTRUCTING THE FILE MAPPING

In Section 3, we defined the *transform* routine. Basically, it takes the *maxline* variables and the *symbol* arrays, and fills out the map defined in Section 2. The function body is shown in Figure 8. The *scan* routines were created to keep the *transform* routine readable. They do this partly by simple smallness. The differences and similarities of the scan loops become more apparent, and the independence of the scratch variables is made explicit. Also, the specifications of the routines document the evolving state of the mapping data, whereas comments within a single large routine tend to be constructed with less care. It may not always be clear just what body of code a comment applies to, a difficulty which routine specifications cannot have.

It should be noted that this program was coded with tab settings at every fifth column. It is well known that an indentation of two columns isn't enough, and that eight is too much. This rule follows from practical experience with large routines. As routines become larger, they need deeper indentation in order to keep groupings visually distinct. On the other hand, deep indentation becomes more likely to run things up against the right margin. This difficulty with size gives us one more reason for keeping routines small, regardless of language.

Another aspect of smallness is economy in the use of lines. There is a practical advantage to fitting an entire routine onto a screen, or onto a page. This program has followed the convention that an opening brace ("curly

```
 * initsymtab        Must be called, once, before any calls to addsymbol.
 * ----------

 * addsymbol    Expects pline-> a string with linelen non-nul chars.
 * ---------    Saves that line into the symbol table.
 *              Returns a handle to the symtab entry for that unique line.
 *              If inoldfile nonzero, then linenum is remembered.
 *              Expects initsymbtab has been called, once.

 * symbolisunique    Arg is a ptr previously returned by addsymbol.
 * --------------    Returns true if the line was added to the
 *                   symbol table exactly once with inoldfile true,
 *                   and exactly once with inoldfile false.

 * lineofsymbol      Arg is a ptr previously returned by addsymbol.
 * ------------      Returns the line number stored with the line.

 * showsymbol        Arg is a ptr previously returned by addsymbol.
 * ----------        Prints the line to stdout.

void initsymtab()

char *addsymbol( pline, linelen, inoldfile, linenum )
char *pline;
int linelen, inoldfile, linenum;

int symbolisunique( psymbol )
char *psymbol;

int lineofsymbol( psymbol )
char *psymbol;

void showsymbol( psymbol )
char *psymbol;
```

FIGURE 7

```
int oldline, newline;
int oldmax = oldinfo.maxline + 2;  /* Count pseudolines at   */
int newmax = newinfo.maxline + 2;  /* ..front and rear of file */

for(oldline=0; oldline < oldmax; oldline++ ) oldinfo.other[oldline]= -1;
for(newline=0; newline < newmax; newline++ ) newinfo.other[newline]= -1;

scanunique();  /* scan for lines used once in both files */
scanafter();   /* scan past sure-matches for non-unique blocks */
scanbefore();  /* scan backwards from sure-matches */
scanblocks();  /* find the fronts and lengths of blocks */
```

FIGURE 8

bracket") is only on a line by itself when starting a function body. However, blank lines have been used to set off groupings, and multi-statement lines have been avoided.

The routines themselves are shown in Figure 9.

7. PRINTOUT

The printing phase essentially scans through the map, printing (or not) the lines that it finds through the map's symbol table handles. This was done with a single loop, which may advance a *newline* variable, or may advance an *oldline* variable, or may advance both. (The advances are always by one, or else by the size of a block.) There are two major problems. The first is simply that there are a large number of cases—for example, if a block has been moved, then a scan may encounter it twice, once where it came from, and once where it went to. The second problem is that the code would have an unreadable control structure if it were written as a single function.

```
/***************************************************************************
 *
 * scanunique   Expects both files in symtab, and oldinfo and newinfo valid.
 * ----------   Scans for lines which are used exactly once in each file.
 *              The appropriate "other" array entries are set to the line# in
 *              the other file.
 *              Claims pseudo-lines at 0 and XXXinfo.maxline+1 are unique.
 *
 ***************************************************************************/
void scanunique()
{
     int oldline, newline;
     char *psymbol;

     for( newline = 1; newline <= newinfo.maxline; newline++ ) {
          psymbol = newinfo.symbol[ newline ];
          if( symbolisunique( psymbol )) {       /* 1 use in each file */
               oldline = lineofsymbol( psymbol );
               newinfo.other[ newline ] = oldline;     /* record a 1-1 map */
               oldinfo.other[ oldline ] = newline;
          }
     }
     newinfo.other[ 0 ] = 0;
     oldinfo.other[ 0 ] = 0;
     newinfo.other[ newinfo.maxline + 1 ] = oldinfo.maxline + 1;
     oldinfo.other[ oldinfo.maxline + 1 ] = newinfo.maxline + 1;
}


/***************************************************************************
 *
 * scanafter    Expects both files in symtab, and oldinfo and newinfo valid.
 * ---------    Expects the "other" arrays contain positive #s to indicate
 *              lines that are unique in both files.
 *              For each such pair of places, scans past in each file.
 *              Contiguous groups of lines that match non-uniquely are
 *              taken to be good-enough matches, and so marked in "other".
 *              Assumes each other[0] is 0.
 *
 ***************************************************************************/
void scanafter()
{
     int oldline, newline;

     for( newline = 0; newline <= newinfo.maxline; newline++ ) {
          oldline = newinfo.other[ newline ];
          if( oldline >= 0 ) {          /* is unique in old & new */
               for(;;) {                /* scan after there in both files */
                    if( ++oldline > oldinfo.maxline   ) break;
                    if( oldinfo.other[ oldline ] >= 0 ) break;
                    if( ++newline > newinfo.maxline   ) break;
                    if( newinfo.other[ newline ] >= 0 ) break;

                    /* oldline & newline exist, and aren't already matched */

                    if( newinfo.symbol[ newline ] !=
                        oldinfo.symbol[ oldline ] ) break;  /* not same */

                    newinfo.other[ newline ] = oldline;   /* record a match */
                    oldinfo.other[ oldline ] = newline;
```

FIGURE 9

```
/*****************************************************************************
*
* scanbefore   As scanafter, except scans towards file fronts.
* ----------   Assumes the off-end lines have been marked as a match.
*
*****************************************************************************/
void scanbefore()
{
     int oldline, newline;

     for( newline = newinfo.maxline + 1; newline > 0; newline-- ) {
          oldline = newinfo.other[ newline ];
          if( oldline >= 0 ) {                /* unique in each */
               for(;;) {
                    if( --oldline <= 0                  ) break;
                    if( oldinfo.other[ oldline ] >= 0 ) break;
                    if( --newline <= 0                  ) break;
                    if( newinfo.other[ newline ] >= 0 ) break;

                    /* oldline and newline exist, and aren't marked yet */

                    if( newinfo.symbol[ newline ] !=
                        oldinfo.symbol[ oldline ] ) break;  /* not same */

                    newinfo.other[ newline ] = oldline;   /* record a match */
                    oldinfo.other[ oldline ] = newline;
               }
          }
     }
}
```

```
/*****************************************************************************
*
* scanblocks   Expects oldinfo valid.
* ----------   Finds the beginnings and lengths of blocks of matches.
*              Sets the blocklen array (see definition).
*
*****************************************************************************/
void scanblocks()
{
     int oldline, newline;
     int oldfront = 0;      /* line# of front of a block in old file, or 0  */
     int newlast = -1;      /* newline's value during the previous iteration*/

     for( oldline = 1; oldline <= oldinfo.maxline; oldline++ )
               blocklen[ oldline ] = 0;
     blocklen[ oldinfo.maxline + 1 ] = UNREAL;    /* starts  a mythical blk */

     for( oldline = 1; oldline <= oldinfo.maxline; oldline++ ) {
          newline = oldinfo.other[ oldline ];
          if( newline < 0 ) oldfront = 0;         /* no match: not in block */
          else{                                   /* match. */
               if( oldfront == 0 )         oldfront = oldline;
               if( newline != (newlast+1)) oldfront = oldline;
               ++blocklen[ oldfront ];
          }
          newlast = newline;
     }
}
```

FIGURE 9. *Continued*

I have chosen to write *printout* as a main function and nine subsidiary functions. They are held together by four global variables, rather than by parameter lists and by result values. This is usually an inferior method, since the use of global variables means that the functions have side effects that in general are hard to document (or are poorly documented). In this specific case, however the subsidiary functions are in fact just fragments of the whole, and the C language makes it burdensome to pass the global variables both in and out of the functions. I apologize for seeming to support a practice which I counsel against.

The variables global to the ten printout functions are shown in Figure 10, and the functions are shown in Figure 11. The reader may have noticed that the *showsame* function contains an error check. It is considered good practice to leave such checks in the final program, unless there are reasons to remove them.

8. FEATURES AND PERFORMANCE

Since the program contains loops that span the inputs, but does not contain any nested loops, we would expect that execution time would be linear in the size of input. In big-oh notation, we would say that we expect

```
enum{ idle, delete, insert, movenew, moveold, same, change } printstatus;
enum{ false, true } anyprinted;
int printoldline, printnewline;          /* line numbers in old & new file */
```

FIGURE 10

```
void printout()
{
     printstatus = idle;
     anyprinted = false;
     for( printoldline = printnewline = 1; ; ) {
          if( printoldline > oldinfo.maxline ) { newconsume(); break;}
          if( printnewline > newinfo.maxline ) { oldconsume(); break;}
          if(      newinfo.other[ printnewline ] < 0 ) {
               if( oldinfo.other[ printoldline ] < 0 )          showchange();
               else                                             showinsert();
          }
          else if( oldinfo.other[ printoldline ] < 0 )          showdelete();
          else if( blocklen[ printoldline ] < 0 )                 skipold();
          else if( oldinfo.other[ printoldline ] == printnewline ) showsame();
          else                                                    showmove();
     }
     if( anyprinted == true ) printf( ">>>> End of differences.0  );
     else                     printf( ">>>> Files are identical.0 );
}

/***************************************************************************
*
* newconsume        Part of printout. Have run out of old file.
* ----------        Print the rest of the new file, as inserts and/or moves.
*
***************************************************************************/
void newconsume()
{
     for(;;) {
          if( printnewline > newinfo.maxline ) break;        /* end of file */
          if( newinfo.other[ printnewline ] < 0 ) showinsert();
          else                                    showmove();
     }
}
```

FIGURE 11

```
/**************************************************************************
 *
 * oldconsume       Part of printout. Have run out of new file.
 * ----------       Process the rest of the old file, printing any
 *                  parts which were deletes or moves.
 *
 **************************************************************************/
void oldconsume()
{
     for(;;) {
          if( printoldline > oldinfo.maxline ) break;       /* end of file */
          printnewline = oldinfo.other[ printoldline ];
          if( printnewline < 0 ) showdelete();
          else if( blocklen[ printoldline ] < 0 ) skipold();
          else showmove();
     }
}

/**************************************************************************
 *
 * showdelete       Part of printout.
 * ----------       Expects printoldline is at a deletion.
 *
 **************************************************************************/
void showdelete()
{
     if( printstatus != delete ) printf( ">>>> DELETE AT %d0, printoldline);
     printstatus = delete;
     showsymbol( oldinfo.symbol[ printoldline ]);
     anyprinted = true;
     printoldline++;
}

/**************************************************************************
 *
 * showinsert       Part of printout.
 * ----------       Expects printnewline is at an insertion.
 *
 **************************************************************************/
void showinsert()
{
     if( printstatus == change ) printf( ">>>>     CHANGED TOO );
     else if( printstatus != insert )
          printf( ">>>> INSERT BEFORE %d0, printoldline );
     printstatus = insert;
     showsymbol( newinfo.symbol[ printnewline ]);
     anyprinted = true;
     printnewline++;
}

/**************************************************************************
 *
 * showchange       Part of printout.
 * ----------       Expects printnewline is an insertion.
 *                  Expects printoldline is a deletion.
 *
 **************************************************************************/
void showchange()
{
     if( printstatus != change )
          printf( ">>>> %d CHANGED FROM0, printoldline );
     printstatus = change;
     showsymbol( oldinfo.symbol[ printoldline ]);
     anyprinted = true;
     printoldline++;
}
```

FIGURE 11. *Continued*

```
/*****************************************************************************
*
* skipold           Part of printout.
* -------           Expects printoldline at start of an old block that has
*                   already been announced as a move.
*                   Skips over the old block.
*
*****************************************************************************/
void skipold()
{
     printstatus = idle;
     for(;;) {
          if( ++printoldline > oldinfo.maxline ) break;     /* end of file  */
          if( oldinfo.other[ printoldline ] < 0 ) break;    /* end of block */
          if( blocklen[ printoldline ]) break;          /* start of another */
     }
}
```

```
/*****************************************************************************
*
* skipnew           Part of printout.
* -------           Expects printnewline is at start of a new block that has
*                   already been announced as a move.
*                   Skips over the new block.
*
*****************************************************************************/
void skipnew()
{
     int oldline;
     printstatus = idle;
     for(;;) {
          if( ++printnewline > newinfo.maxline ) break;    /* end of file  */
          oldline = newinfo.other[ printnewline ];
          if( oldline < 0 ) break;                         /* end of block */
          if( blocklen[ oldline ]) break;              /* start of another */
     }
}
```

```
/*****************************************************************************
*
* showsame          Part of printout.
* --------          Expects printnewline and printoldline at start of
*                   two blocks that aren't to be displayed.
*
*****************************************************************************/
void showsame()
{
     int count;
     printstatus = idle;
     if( newinfo.other[ printnewline ] != printoldline ) {
          fprintf( stderr, "BUG IN LINE REFERENCING07" );     /* (bel) */
          exit(1);
     }
     count = blocklen[ printoldline ];
     printoldline += count;
     printnewline += count;
}
```

FIGURE 11. *Continued*

```
/*****************************************************************************
 *
 * showmove          Part of printout.
 * --------          Expects printoldline, printnewline at start of
 *                   two different blocks ( a move was done).
 *
 *****************************************************************************/
void showmove()
{
     int oldblock = blocklen[ printoldline ];
     int newother = newinfo.other[ printnewline ];
     int newblock = blocklen[ newother ];

     if( newblock < 0 ) skipnew();          /* already printed */
     else if( oldblock >= newblock ) {      /* assume new's blk moved */
          blocklen[ newother ] = -1;        /* stamp block as "printed" */
          printf( ">>>> %d MOVED TO BEFORE %d0, newother, printoldline );
          for( ; newblock > 0; newblock--, printnewline++ )
               showsymbol( newinfo.symbol[ printnewline ]);
          anyprinted = true;
          printstatus = idle;
     }else                            /* assume old's block moved */
          skipold();                  /* target line# not known, display later */
}
```

FIGURE 11. *Continued*

execution to be $O(N)$, where N is the number of input lines. This analysis assumes that relatively little is printed out, since that is the usual case. This analysis also ignores the presence of the binary tree used by the symbol table package. Since the size of this tree is $O(U)$, where U is the number of unique lines, we can expect the tree construction phase to have an execution time of $O(N \log_2(U))$.

This program uses a fixed amount of space for the map. The original, more complicated version used $O(N)$ space, with some loss in both simplicity and speed. (The symbol arrays were implemented as arrays of pointers to arrays, with dynamic allocation of the subarrays as needed during the input phase. The *other* arrays and the *blocklen* array need not be allocated until after the input phase, at which time the desired size is known exactly.) (The original program is also capable of keeping references into the input files, rather than keeping the actual lines themselves. This greatly shrinks the symbol table, but will give incorrect results should a hash collision occur.)

To measure this specific program's performance, I constructed several large (>10,000-line) input files, and for each, I constructed a version of it which differed slightly. I compiled the program, with optimization requested, and timed it on these input files, using the *time* command of a Sun-3/160 workstation. The program took approximately 25 percent to 50 percent longer than the standard *diff* utility of this machine. The *gprof* profiling tool revealed that the *transform* step was taking 2.6 percent of the execution time, and the printout step was taking less than 0.1 percent of the time. This indicates that the code for these steps is in no need of performance tuning, and no effort should be wasted on attempts to improve their speed. Of course, this conclusion depends on several "reasonable" assumptions. (For example, the speed of the input phase is affected by average line length, whereas the speed of the transform step is not.)

The symbol table package (not presented in this column) is clearly inefficient, with addsymbol consuming 60 percent of the execution time. This is due to its simplistic algorithm, which does a full string comparison at every step of a tree descent. There are several ways to reduce this cost. As noted in a previous section, the strings can be shortened by a compression method. Comparison can be avoided when strings are of unequal length. The tree depth can be minimized by a balancing method. A hashing technique may be used instead of a tree. Or, the hash of each line may be carried around with the line, so that the bulk of the comparisons can be done on the hash values. This last technique was coded into a version of the program, and the execution time became comparable to that of the *diff* utility.

The quality of the algorithm's decisions was discussed in the article by Heckel. To summarize, the output is usually of a quality comparable to that of other algorithms. Sometimes the output is "more right," particularly because it is capable of noticing a block move as such, rather than noticing it as a block deletion and as a (separate) block insertion. There are inputs which will cause the algorithm to make poor decisions: this can also be said of the other major algorithms. The failures are often a consequence of the fact that files may contain many identical lines, particularly if they are program source. Each algorithm must resolve this ambiguity, and there may in fact be no resolution which is "right." In general, however, this algorithm does produce the "right" result.

Historically, file difference programs have been subject to enhancement. One main category of changes has been in the area of input filtering. This is usually optional low-level processing, such as case reduction, various forms of whitespace reduction, comment stripping, and the like. Another category is optional changes in printout format, to show the context of a change, or to be more suitable for some other tool, such as an editor or a revision control system. A more open-ended category is changes made to fit the program into some system context. This may involve adding knowledge of some structured environment (such as hierarchical directories), or may involve adaptation to ideas such as "uninteresting changes," such as the timestamps found in regression-test logs. The program presented here has been designed and coded in a manner which should make it suitable for maintenance, and therefore a reasonable platform for enhancements.

Acknowledgment. This program would never have been written if the original exposition by Paul Heckel had not been so persuasive. I would like to thank my colleagues at Symbionics, for whom the original implementation was written. I would also like to thank my colleagues on the Archons project at Carnegie-Mellon University, whose support and facilities were essential to writing this column.

Donald C. Lindsay
Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

A Review of Donald C. Lindsay's Text File Difference Utility, *diff*

Harold Thimbleby is Professor of Information Technology at Stirling University. He was awarded the British Computer Society Wilkes Medal for his paper on literate programming [2].

Overview

So far, all reviews of literate programs in *Communications* have criticized the content rather than the use of literate programming itself. This suggests that literate programming successfully brings out details of content, and makes programs clearer.

In my review of Donald Lindsay's *diff* program I shall comment on his use of literate programming as a method, rather than the content of his particular program.

I shall first make some rather general and abstract comments about literate programming, then make some particular comments about Lindsay's use of literate programming.

Lindsay appears to have simulated literate programming: that is, he has generated the sort of outcome one would expect, but obtained by a manual method. What would Lindsay's program have been like if he had been able to get the same, or better, results without effort? I had a literate programming system available, so I was able to rekey his program to compare manual and automatic methods. This is an unconventional way to review, but in fact, I did what any programmer would do faced with the task of converting a conventional, though heavily documented, program into a literate program. My conclusions are quite general. I claim that literate programming not only facilitates program improvement, but actually encourages it, for, as we shall see, the beneficial facilities are free.

Of Literate Programming and Programming Paradigms

The term *programming paradigm* is now widely but imprecisely used. I shall define the term as follows so that I can talk about "a literate programming paradigm" as a useful concept.

Definition: A programming paradigm is the set of features that a programming system provides for free, and with warranty that such features are correct (and generally efficient).

For example, backtracking is provided for free in Prolog, but a Pascal programmer has to try very hard to get backtracking. Thus being able to assume the presence of backtracking is part of the paradigm of Prolog, but not of Pascal. Motivated Pascal programmers can, of course, still do backtracking, but the effort and unreliability of doing it usually discourages them.

Now, consider literate programming. If literate programming is to be "paradigmatic" then it must provide features for free, and it must provide a warranty. That way, programmers will be able to—and want to—take advantage of the features. The effect of the freedom and assurance should be to liberate programmers to concentrate on their pressing programming problems, yet almost unconsciously take advantage of the paradigm. In a good literate programming environment, the trappings of literateness should flow as if the programmer was an accomplished expositor—just as a Prolog programmer can very easily do things that, in Pascal, take accomplished programming.

REVIEW

Before I embark on my criticism, I want to say that I admire the author for his courage in presenting his program to the public. And it is good that literate programming is such an attractive medium that we can assume many people will be interested in reading and scrutinizing programs that would otherwise be consigned to obscurity.

Lindsay has presented a text file difference utility, *diff*. It is based on a real program, written for real use, though now simplified for presentation. Lindsay intimates that we may find his style too cryptic for the purposes of presentation, but perhaps only in comparison with a textbook style exposition with its pedagogic tendencies. It must be said, however, that it is difficult to reconcile the style of commentary encouraged by literate programming with the needs of different sorts of readers: the commentary for a published program is different in nature than the commentary needed by a program maintainer, or, indeed, the program writer.

So far as I can see, Lindsay's literate program started out as a conventional program, then was somewhat edited, and *then* interleaved with new commentary. It appears that the new commentary is mostly about the process of programming and general design issues, rather than about the program itself. Almost all of the program documentation remains in a purely conventional style, in standard comments.

Lindsay has had to work, and probably work very hard, to get the final effect. Evidently, this transformation was achieved by hand.

Although there is some explicit cross-referencing, much of it would have been made redundant by automatic cross-referencing; the rest would have been systematized. Other literate programming effects I consider desirable are omitted altogether: indeed, it would have been hard, and unreliable, work to do them by hand. For example, there is no index. All in all, this is in contrast with what we would expect had literate programming been available as an effective paradigm. The order of the program is apparently unchanged from the original code, even though literate programming freely permits an arbitrary code order to simplify exposition. Of course, it would be much easier to change the code order, and keep the result intelligible, if there was automatic cross-referencing.

But it must be emphasized that only *exceedingly* good literature can be recognized as such from small fragments (unless the author has a reputation). This is a problem for the review of such small programs as can be presented in *Communications.* Yet, if the paraphernalia of literature came free and correct, one would tend to be influenced by them. We would expect a literate program to be quite liberal with such paradigmatic features as: flexible order of elaboration, cross-referencing, indices, typographical niceties, mnemonic names.

So we have a program edited *by hand* for presentation, with various elisions (the code that is presented is not compilable as it stands). What assurance do we have that this *is* the actual program? None. There may have been clerical errors made in the transformation from compilable program to literate program. This is a most serious criticism—and, conversely, an advert for literate programming done automatically.

Of course, this is (or was originally) a real program and presumably implemented in a regime that did not provide a literate programming environment. Has Lindsay emulated literate programming given such restriction? I fear not. One of the most persuasive paradigmatic features of literate programming is that *exactly* the document you are reading can be mechanically processed to obtain the program. This is a warrant of the paradigm.

Literate programming encourages a programmer to elaborate his program with documentation, and presents the program nicely, in a form conducive to reading. These should be a single source document (or file) containing *both* documentation and program. Program and documentation can be developed concurrently *in the same place*, without overhead (this is part of the paradigm: a feature that comes "for free"). But in the program under review we find a text which (apparently) was developed first as a conventional program, then edited, then documented (or rather used as a vehicle to carry certain textbook-style comments). There is no way we can expect any programmer to develop both program and documentation concurrently with such a struggle. What would happen, if in the process of documenting a fragment of program, the author realized there was an opportunity to improve the program. Would he go over the whole process again? Surely not. The effort put into transforming a program by hand represents a commitment that will not be readily undone. Modifying a program squanders earlier effort put into preparing it for presentation. But in a literate programming environment, the process would be paradigmatic: it would cost the programmer *nothing* to change the program as soon as he noticed any opportunity for improvement. That way we would get better programs, faster.

An Experiment . . .

There is no need to continue criticizing the program, once the point has been made. I understand the constraints and the desire to present a real program.

As an experiment, and in hypothetical support of my claims, I rekeyed Lindsay's program together with his documentation. I used **cweb**, a literate programming system I developed in 1983 [2].

At first, apart from ignoring meta-documentation (that one would not normally expect to find in a program outside of a textbook), I did not edit his text in any way, except to take advantage of the literate programming paradigm. The changes I did make (mainly entering section delimiters at the right moment) were an insignificant part of copy-typing the text. I typed 1.25 percent extra in order to satisfy the conventions of my system, plus 1.89 percent (beyond what I could actually see) for typographical niceties, such as arranging for in-line comments to be vertically aligned.[1] In comparison, I estimate that Lindsay's source contains maybe a 5 percent overhead in the way of formatting commands (but I can only guess what formatter he used).

I made a few changes subsequently (e.g., improving the order of presentation; promoting in-line comments to separate documentation), but these could charitably be counted a normal part of proofreading, a task which was in any case required to check my copy-typing against the original. **Cweb** itself imposes no restrictions on one's programming, and there was no *need* for me to make any changes whatsoever. Tempted by the paradigm, however, I succumbed to unfaithful copy-typing.

As an experiment to compare the program in original and as-it-were paradigmatized forms, it was bad experimental method to *improve* the program, but it emphasizes the point. In contrast, Lindsay apparently *simplified* the program; this may be due to the effort of simulating a literate style. If so, then this would be an indictment of simulating literate programming. The original program may have been simplified for other reasons, nothing to do with literate programming: few real programs are suitable for direct publication, since they are typically too long and too machine- and environment-specific.

For reasons of space (and since I basically copy-typed the program) it is not necessary to present the literate version of Lindsay's program. The general effect of literate programming, and the widespread use of the features I have mentioned, can be discerned in, for instance, Knuth's books [1].

[1] **cweb** requires explicit commands for structuring a program. Had I had an interactive literate programming system or one using grammar-directed structuring, such as Welsh et al.'s [3], there need have been no overhead.

Looking at the result of my rekeying, I was surprised how the original commentary (which looked all right embedded in code) looked insubstantial when set apart in the literate style. Of course, I used the original commentary in a way that may not have been intended, and which must give an unfair impression. Lindsay would no doubt want to improve it. In general, this effect of literate programming (making commentary more prominent) would encourage even better documentation.

Summary

Returning to the programming paradigm idea: it is not so much what you do (for you can do the same things in any programming language if you try hard enough), but how you do it, and how easily. A literate programming style is not, to my mind, what literate programming is all about. How literate programming is done, and how easily it can be done and redone, changes the way one programs. It provides new incentives. There is an incentive to make code and documentation consistent (by developing code and documentation concurrently). There is an incentive to explain, and hence understand what you are doing. And by making a program look so nice, it gives an incentive to publicize the program and suffer its public review! In the future, I look forward to the time when programmers are so encouraged that they feel able to distribute *real* programs in source form, including their literate documentation.

Harold Thimbleby
Department of Computing Science
University of Stirling
Stirling FK9 4LA, Scotland

The reviewer, Harold Thimbleby, adds:

I am glad to see that Lindsay's program has been improved since I reviewed it, but I am embarrassed that some of my comments now seem inappropriate. However, useful lessons may be drawn out of this experience.

1. There are a number of minor changes. Just one example: my review says that Lindsay intimates that we may find his style too cryptic. That was his original word.
2. There are more substantial changes. For example, Lindsay has added. "This column should not be taken as a 'literate program,' in Knuth's restricted sense . . ." With or without this explicit claim, the material is now quite clearly a standard commentary plus fragments of program. As such I would have had difficulty reviewing it as a literate program.
3. The original manuscript contained interleaved commentary and program, in the style of literate programming. In the process of publication, all the program text has been separated out *by the printers* as numbered figures. I must emphasize my review's comments about warranties: we have no assurance that this numbering is correct, and it would surely go wrong with more than Lindsay's 11 figures taken from his cut-down program. The original program cannot be reconstructed from such sparse representation.
4. The proofs I was sent to check did not include any program code. It is ironic that literate programming aims to combine documentation and code so that they may be created, checked and published together. In the present case, the process of publication has completely separated them and any correspondence cannot be checked.

Indeed, the code I originally reviewed was poor and I deferred to supply only positive comments drawn out of Lindsay's attempt at emulating literate programming, however he actually chose to program. Now, I have no idea if the code has improved and whether my judgment would still be appropriate.

5. My section "An experiment . . .", particularly detailed comparisons, must be taken to refer precisely to the manuscript I reviewed. The general comments of this section stand.

In summary: on the one hand, although it is perfectly natural for Lindsay to respond to criticism (e.g., adding his comment about a machine-generated index), it is regrettable that in some details the review now appears inaccurate; on the other hand, the changes that have been made to the original program—the effort, omissions and concomitant risk of error brought about by conceding to the pressures of publication—emphasise the great advantages of doing literate programming automatically, paradigmatically, properly.

REFERENCES

1. Knuth, D.E. *Computers and Typesetting*, e.g., Volumes **B** & **D**, Addison-Wesley, Reading, Mass., 1986.
2. Thimbleby, H.W. Experiences of 'Literate Programming' using **cweb** (a variant of Knuth's WEB), *Comp. J. 29*, 3 (June 1986), 201–211.
3. Welsh, J., Rose, G.A., & Lloyd, M. An Adaptive Program Editor. *Australian Comp. J. 18*, 2 (May 1986), 67–74.

For Correspondence: Christopher J. Van Wyk. AT&T Bell Laboratories, Room 2C-457, 600 Mountain Avenue, Murray Hill, NJ 07974.