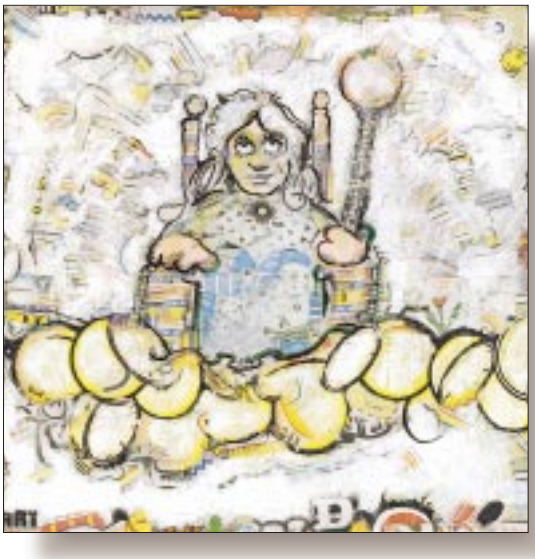
by Jeffreys Copeland and Haemer



Comparing Text, Part 1

Jeffrey Copeland *(*`copeland@alumni.caltech.edu`*) is at Softway Systems Inc. in Boulder, CO, working on UNIX internationalization. He spends his spare time rearing children, raising cats, and being a thorn in the side of his local school board.*

Jeffrey S. Haemer *(*`jsh@usenix.org`*) works at QMS Inc. in Boulder, CO, building laser printer firmware. Before he worked for QMS, he operated his own consulting firm, and did a lot of other things, like everyone else in the software industry.*

Note: The software from this column is available at `http://alumni.caltech.edu/~copeland/work.html`.

You're all familiar with the `diff` utility. It's one of the more powerful tools we have for keeping track of what's changed in a text file. Over this past summer and fall, we have been working on a series of documents in different stages of completion, with different revisions, and we've realized that there's a misfeature in the operation of `diff`: If we are trying to compare formatted versions of a document rather than the document source, `diff` gives us thousands of lines of spurious differences.

To steal a short example from the Free Software Foundation's GNU `diff` documentation, consider two files, `lao` and `tzu`, with slightly different formatting:

```
$ cat lao
 The Way that can be told of is not the eternal Way;
 The name that can be named is not the eternal name.
 The Nameless is the origin of Heaven and Earth;
 The Named is the mother of all things.
 Therefore let there always be non-being,
 so we may see their subtlety,
 And let there always be being,
 so we may see their outcome.
 The two are the same,
 But after they are produced,
 they have different names.
$ cat tzu
 The Nameless is the origin of Heaven and Earth;
 The named is the mother of all things.

 Therefore let there always be non-being, so we may see their subtlety,
 And let there always be being, so we may see their outcome.
 The two are the same,
 But after they are produced, they have different names.
 They both may be called deep and profound.
 Deeper and more profound, The door of all subtleties!
```

Even though the text beginning with `Therefore...` is identical in both files, the remainder of the file is reported by `diff` as different because the formatting is different. Not even the useful `-b` or `-w` flags, which are used to ignore white space, can help us. What's worse, if we only have Microsoft Corp. Word files, we normally extract the text with a line like:

```
strings foo.doc | fmt -75 >foo.txt
```

This gives us a reasonable ASCII file to peruse, so we don't have to rely on the Microsoft tool. Unfortunately, this is the equivalent of a formatted document, so we're back to square one and unable to compare the old text with a new version. (OK, we'll concede that if we were willing to use one of the what-you-see-is-what-you-get word processors, we could use its "redlining" feature to mark the differences for us.)

We could delve into the source code of `diff` and add a new flag, or we could retreat to the original papers on `diff` and write something from scratch. (The papers are: "An O(ND) Difference Algorithm and its Variations," by Eugene W. Myers, *Algorithmica* , Vol. 1 (1986), pp. 251-266; and "A File Comparison Program," by Webb Miller and Eugene W. Myers, *Software–Practice and Experience*, Vol. 15 (1985), pp. 1025-1040.) Alternatively, we could graft a filter onto the input or output of `diff` to remove the spurious text differences.

We considered each of these approaches. Writing a new program from scratch struck us as the most interesting, though doing all the work inside the `diff` source code is probably more practical because it wouldn't necessitate rewriting the code for traversing the difference tree. However, in the interest of getting something to work in the short term, we opted for the third approach and built a shell script to achieve our desired output.

Why do it this way? It's a time-honored prototyping technique for UNIX tools. Remember that the original version of `spell` was a half-dozen-line shell script. Exercise for the reader: Given the prototype we're providing, either write a new program or modify the GNU `diff` source to do the function we are creating here.

A Shell Script

We need a way to compare the running text without having the line breaks get in the way. We can do this by breaking each line from each file into individual words, using a command like `fmt -l 2 foo.txt >foo.words`, and then comparing the two files of words against each other. Thus, our first cut at a script called `redline` might be something like:

```
#! /bin/sh
fmt -l 2 $1 >/tmp/$$a
fmt -l 2 $2 >/tmp/$$b
diff -b /tmp/$$a /tmp/$$b
```

We're deliberately not doing error checking or cleanup yet–we're just trying to prove the concept–we'll get to a properly constructed shell script shortly. Also, you might have to modify those command lines. If your system is based on Free BSD, for example, the `fmt` command line would be `fmt 2 $1`. This gives us output that begins:

```
2,24d1
<         Way
<         that
<         can
<         be
<         told
<         of
<         is
<         not
<         the
<         eternal
<         Way;
<         The
<         name
<         that
```

That output is not of much use: It presents us with a difference output consisting of one word per line, which, while correct, is difficult to read. Further, we still don't have context for the differences because they appear in a vacuum.

We can improve the situation by generating a side-by-side difference of the word lists, like this:

```
#! /bin/sh
fmt -l 2 $1 >/tmp/$$a
fmt -l 2 $2 >/tmp/$$b
diff -y -b /tmp/$$a /tmp/$$b
```

Which gives us a full context, beginning like so:

```
The           The
Way       <
that      <
can       <
be        <
told      <
of        <
is        <
not       <
the       <
eternal   <
Way;      <
The       <
name      <
that      <
can       <
be        <
named     <
is        <
not       <
```

Note: If you're not using the GNU `diff`, then you may need to use the `sdiff` command.

We have the full context again, but it's still painful to read.

Unfortunately, we can't do something as simple as pipe the text back through `fmt`, because the difference markers will be folded into the formatted text, which will make it even harder to read and make the differences even more difficult to spot. We really want to be able to postprocess the output of `diff` in some way to make the output easier to read.

Also, notice that we've had to use the `-b` flag to `diff` to prevent spurious differences caused by the differing number of blanks at the beginning of lines. We should probably remove all the blanks at the beginning of lines to prevent this. However, this means that we need to add a blank line between indented paragraphs too. With that in mind, and postulating a postprocessing filter named `reddiff`, our next cut at the shell script looks something like this:

```
#! /bin/sh

#  begin by breaking files into a word per line;
#   ensure that paragraphs are handled nicely
#   whether they're indented or preceded by
#   blank lines
expand $1 | sed -e 's/^  */\
/' | fmt -2 >/tmp/$$a
expand $2 | sed -e 's/^  */\
/' | fmt -2 >/tmp/$$b

#  now do an sdiff, and collect the differences
diff -y /tmp/$$a /tmp/$$b | reddiff
```

We'd normally have used a character class in the `sed` regular expression that matched either space or tab, but instead we used `expand` to convert tabs to spaces in this example because it's easier to see what the code is doing–feel free to fix this in your version. In the best of all possible worlds, we'd be able to use POSIX character classes in our `sed` expressions, and use a line such as `sed 's/[:blank:]*//'`.

That leaves us with two tasks: First, we must add file cleanup and error checking to the script; second, we must write the `reddiff` program. Let's do the easier task first.

We begin by providing a description of the program, and an RCS identification string. We follow the commentary by checking the file arguments and issuing a usage message if one is needed. Next, we set the cleanup of temporary files through the use of a shell `trap` command. For those of you not familiar with it, `trap` executes the given code when any of the named signals are received by the script–an older version required you to provide the signal *number*, which made it fairly nonportable. Then, we proceed with the code as outlined before:

```
#! /bin/sh
# $Id: $
#  This does a diff on running text,
#   in the same style as a Word or
#   WordPerfect red line comparison.

#  set the cleanup
trap 'rm -f /tmp/$$*'  EXIT HUP QUIT INT TERM

#  check that file arguments are present
[ -z "$1" -o -z "$2" ]  &&
  echo usage:  $0 file1 file2  &&
  exit

#  begin by breaking files into a word per line
#  ensure that paragraphs are handled nicely
#  whether they're indented or preceded by
#  blank lines
expand $1 | sed -e 's/^  */\
/' | fmt -2 >/tmp/$$a
expand $2 | sed -e 's/^  */\
/' | fmt -2 >/tmp/$$b

#  now do an sdiff, and collect the differences
diff -y /tmp/$$a /tmp/$$b | expand | reddiff
```

Because `diff -y` produces tabs as part of its white space on output, we're expanding those tabs to make parsing by the `reddiff` filter easier.

The next task is a little more complicated.

The Postprocessing Filter

We can make our development task easier by capturing a sample of the intended input to `reddiff` for testing purposes. We simply substitute `cat` into the `redline` script in place of `reddiff`. The logical first routine to provide for our `reddiff` program is one to parse the output of `diff -y`, and return the words. There are four possible forms to a line of output from `diff -y`. We can have the input be identical:

```
The         The
```

The input can be changed between the files:

```
Named     | named
```

The line can appear in the first file only:

```
that      <
```

The line can appear in the second file only:

```
          > called
```

All of the example outputs have possible text, a tag character and possible text. In this case, "possible text" represents a single word:

```
/* parse the actual output of sdiff */
char
parse( char *line, char **wp1, char **wp2 )
{
```

We provide the line of input itself, and pointers to locations

where the words should be returned. The actual value returned by the function is the `tag` character. We keep the words on the input line as local `static char` pointers. Also, we need some local variables:

```
static char *word1, *word2;
char *s, tag;
```

If we have a completely blank line, it represents a paragraph break, so we return an empty tag and new lines for the words:

```
/* a completely blank line */
if( *line == '\n' )
{
  *line = 0;
  word1 = word2 = line;
  *wp1 = *wp2 = word1;
  return ' ';
}
```

In the normal course of events, though, the `tag` is at a fixed position on the line–we've been compressing the blanks in the example `sdiff` lines we've been showing you; the default line is much wider. We get pointers to the two words:

```
tag = line[62];
word1 = &line[0];
word2 = &line[64];
```

We also need to put `NULL`s at the end of the words, so they can be used as strings:

```
if( (s=strpbrk(word1, " ")) != NULL )
  *s = 0;
 else
  line[21] = 0;
if( (s=strpbrk(word2, "\n")) != NULL )
  *s = 0;
else
     line[84] = 0;
```

Notice that we're doing some defensive programming for words that don't have a terminating blank.

Last, we stuff the pointers to the words into the return locations and return:

```
 *wp1 = word1;
 *wp2 = word2;
 return tag;
}
```

That's all we have time for now. We'll finish showing you the code for the rest of the `reddiff` program next month as our New Year's gift.

Until then, happy holidays and happy trails.