SPIFF -- A Program for Making Controlled Approximate Comparisons of Files

Daniel Nachbar

Software Engineering Research Group
Bell Communications Research
Morristown, New Jersey

ABSTRACT

The well known program **diff** [1] is inappropriate for some common tasks such as comparing the output of floating point calculations where roundoff errors lead **diff** astray and comparing program source code where some differences in the text (such as white space and comments) have no effect on the operation of the compiled code. A new program, named **spiff,** addresses these and other similar cases by lexical parsing of the input files and then applying a differencing algorithm to the token sequences. **Spiff** ignores differences between floating point numbers that are below a user settable tolerance. Other features include user settable commenting and literal string conventions and a choice of differencing algorithm. There is also an interactive mode wherein the input texts are displayed with differences highlighted. The user can change numeric tolerances "on the fly" and **spiff** will adjust the highlighting accordingly.

Some Troubles With Diff

Over the past several years, it has been fairly easy to tell when a new type of computer arrived at a nearby computer center. The best clue was the discordant chorus of groaning, sighing, gnashing of teeth, pounding of foreheads on desks, and other sounds of distress. Tracing these noises to their source, one would find some poor soul in the process of installing a numerical analysis package on the new machine.

One might expect that "moving up" to a new machine would be a cause for celebration. After all, new machines are typically bigger, faster, and better than old machines. However, the floating point arithmetic on any new machine is frequently slightly different from any old machine. As a consequence, software package test routines produce output that is slightly different, but still correct, on the new machines. Serious troubles appear when the person installing the software package attempts to compare the test output files from two different machines by using a difference finding program such as **diff.** Programs such as **diff** do a character by character comparison. **Diff** finds a great many differences, most of which are due to roundoff errors in the least significant digits of floating point numbers. Others are the result of differences in the way in which the two test runs had printed a number (3.4e-1 vs. 0.34). In one case, the test suite for the S statistical analysis package[2], over 1700 floating point numbers are produced (per machine). In the eyes of **diff,** roughly 1200 of these numbers are different. However, none of the "differences" are important ones. Nonetheless, software installers wind up inspecting the output by eye.

A similar problem arises when one attempts to look for differences between two versions of the same C program. **Diff** reports many differences that are not of interest. In particular, white space (except inside quotation marks) and anything inside a comment have no effect on the operation of the compiled program and are usually not of interest. **Diff** does have a mode of operation where white space within a line (spaces and tabs) can be ignored. However, differences in the placement of

newlines cannot be ignored. This is particularly annoying since C programming styles differ on whether to place a newline character before or after the '{' characters that start blocks.

The Problem in General Terms

As already mentioned, programs such as **diff** do a character-by-character comparison of the input files. However, when it comes to interpreting the contents of a file (either by a human or by a program) it is almost never the case that characters are treated individually. Rather, characters make up tokens such as words and numbers, or act as separators between these tokens. When comparing files, one is usually looking for differences between these tokens, not the characters that make them up or the characters that separate them.

What is needed is a program that first parses the input files into tokens, and then applies a differencing algorithm to the token sequences. In addition to finding differences in terms of tokens, it is possible to interpret the tokens and compare different types of tokens in different ways. Numbers, for example, can differ by a lot or a little.[1] It is possible to use a tolerance when comparing two number tokens and report only those differences that exceed the tolerance.

Design Issues

A serious design issue for such a program is how complex to make the parse. The *deeper* one goes in the parsing the larger the unit of text that can be manipulated. For instance, if one is looking for differences in C code, a complete parse tree can be produced and the differencing algorithm could examine insertion and deletion of entire branches of the tree. However, deep parsing requires much more complex parsing and slower differencing algorithms.

Another design issue is deciding how to interpret the tokens. Closer interpretation may lead to greater flexibility in comparing tokens, but also results in a more cumbersome and error-prone implementation.

In the program described here, we attempt to keep both the depth of the parse and the semantics of the tokens to a minimum. The parse is a simple lexical parse with the input files broken up into one dimensional sequences of numbers, literal strings and white space. Literal strings and white space are not interpreted. Numbers are treated as representing points on the real number line.

Default Operation

Spiff[2] works very much like **diff.** It reads two files, looks for differences, and prints a listing of the differences in the form of an edit script.[3] As already suggested, **spiff** parses the files into literal strings and real numbers. The definition of these tokens can be altered somewhat by the user (more on this later). For now, suffice it to say that literals are strings like "cow", "sit", "into", etc. Real numbers look like "1.3", "1.6e-4" and so on. All of the common formats for real numbers are recognized. The only requirements for a string to be treated as a real number is the presence of a period and at least one digit. By default, a string of digits without a decimal point (such as "1988") is not considered to be a real number, but rather a literal string.[4] Each non-alphanumeric character (such as #$@^&*) is parsed into a separate literal token.

Once **spiff** determines the two sequences of tokens, it compares members of the first sequence with members of the second sequence. If two tokens are of different types, **spiff** deems them to be different, regardless of their content. If both tokens are literal tokens, **spiff** will deem them to be

---

[1] Current differencing programs do not have such a notion because the difference between two characters is a binary function. Two characters are the same or they are not.

[2] We picked the name as a way to pay a small tribute to that famous intergalactic adventurer Spaceman Spiff[3]. **Spiff** is also a contraction of "spiffy diff".

[3] An edit script is a sequence of insertions and deletions that will transform the first file into the second.

[4] Integer numbers are often used as indices, labels, and so on. Under these circumstances, it is more appropriate to treat them as literals. Our choice of default was driven by a design goal of having **spiff** be very conservative when choosing to ignore differences.

different if any of their characters differ. When comparing two real numbers, **spiff** will deem them to be different only if the difference in their values exceeds a user settable tolerance.

Altering Spiff's Operation

To make **spiff** more generally useful, the user can control:

- how text strings are parsed into tokens
- how tokens of the same type are compared
- the choice of differencing algorithm used
- and the granularity of edit considered by the differencing algorithm.

These features are described next.

Altering the Parse

The operation of the parser can be altered in several ways. The user can specify that delimited sections of text are to be ignored completely. This is useful for selectively ignoring the contents of comments in programs. Similarly, the user can specify that delimited sections of text (including white space) be treated as a single literal token. So, literal strings in program text can be treated appropriately. Multiple sets of delimiters may be specified at once (to handle cases such as the Modula-2 programming language where there are two ways to specify quoted strings). At present, the delimiters must be fixed string (possibly restricted to the beginning of the line) or end of line. As a consequence of the mechanism for specifying literal strings, multicharacter operators (such as the += operator in C) can be parsed into a single token.

As yet, no provision is made for allowing delimiter specification in terms of regular expressions. This omission was made for the sake of simplifying the parser. Nothing prevents the addition of regular expressions in the future. However, the simple mechanism already in place handles the literal string and commenting conventions for most well known programming languages.[5]

In addition to controlling literal string and comments, the user may also specify whether to treat white space characters as any other non-alphanumeric character (in other words, parse each white space character into its own literal token), whether to parse sign markers as part of the number that they precede or as separate tokens, whether to treat numbers without printed decimal markers (e.g. "1988") as real numbers rather than as literal strings, and whether to parse real numbers into literal tokens.

Altering the Comparison of Individual Tokens

As mentioned earlier, the user can set a tolerance below which differences between real numbers are ignored. **Spiff** allows two kinds of tolerances: absolute and relative. Specifying an absolute tolerance will cause **spiff** to ignore differences that are less than the specified value. For instance, specifying an absolute tolerance of 0.01 will cause only those differences greater than or equal to 0.01 to be reported. Specifying a relative tolerance will cause **spiff** to ignore differences that are smaller than some fraction of the number of larger magnitude. Specifically, the value of the tolerance is interpreted as a fraction of the larger (in absolute terms) of the two floating point numbers being compared. For example, specifying a relative tolerance of 0.1 will cause the two floating point numbers 1.0 and 0.91 to be deemed within tolerance. The numbers 1.0 and 0.9 will be outside the tolerance. Absolute and relative tolerances can be OR'ed together. In fact, the most effective way to ignore differences that are due to roundoff errors in floating point calculations is to use both a relative tolerance (to handle limits in precision) as well as an absolute tolerance (to handle cases when one number is zero and the other number is almost zero).[6] In addition, the user can specify an infinite tolerance. This is useful for checking the format of output while ignoring the actual numbers produced.

---

[5] See the manual page in the appendix for examples of handling C, Bourne Shell, Fortran, Lisp, Pascal, and Modula-2. The only cases that are known not to work are comments in BASIC and Hollerith strings in Fortran.

[6] All numbers differ from zero by 100% of their magnitude. Thus, to handle numbers that are near zero, one would have to specify a relative tolerance of 100% which would be unreasonably large when both numbers are non-zero.

Altering the Differencing Algorithm

By default, **spiff** produces a minimal edit sequence (using the Miller/Myers differencing algorithm[4]) that will convert the first file into the second. However, a minimal edit sequences is not always desirable. For example, for the following two tables of numbers:

0.1 0.2 0.3	0.2 0.3 0.4
0.4 0.5 0.6	0.5 0.6 0.7

a minimal edit sequence to convert the table on the left into the table on the right be to would delete the first number (0.1) and insert 0.7 at the end.[7] Such a result, while logically correct, does not provide a good picture of the differences between the two files. In general, for text with a very definite structure (such as tables), we may not want to consider insertions and deletions at all, but only one-to-one changes.[8] So, rather than look for a minimal edit script, we merely want to compare each token in the first file with the corresponding token in the second file.

The user can choose which differencing algorithm to use (the default Miller/Myers or the alternative one-to-one comparison) based upon what is known about the input files. In general, files produced mechanically (such the output from test suites) have a very regular structure and the one-to-one comparison works surprisingly well. For files created by humans, the Miller/Myers algorithm is more appropriate. There is nothing in **spiff's** internal design that limits the number of differencing algorithms that it can run. Other differencing algorithms, in particular the one used in **diff,** will probably be added later.

Altering the Granularity of the Edit Sequence

By default, **spiff** produces an edit sequence in terms of insertions and deletions of individual tokens. At times it may be more useful to treat the contents of the files as tokens when looking for differences but express the edit script in terms of entire lines of the files rather than individual tokens.[9] **Spiff** provides a facility for restricting the edits to entire lines.

Treating Parts of the Files Differently

For complex input files, it is important that different parts of the file be treated in different ways. In other words, it may be impossible to find one set of parsing/differencing rules that work well for the entire file. **Spiff** can differentiate between parts of the input files on two bases: within a line and between lines. Within a line, a different tolerance can be applied to each real number. The tolerances are specified in terms of the ordinal position of the numbers on the line (i.e. one tolerance is applied to the first real number on each line, a different tolerance is applied to the second number on each line, a third tolerance is applied to the third, and so on). If more numbers appear on a line than there are tolerances specified, the last tolerance is applied to all subsequent numbers on the line (i.e., if the user specifies three tolerances, the third is applied to the third, fourth fifth, . . . number on each line). This feature is useful for applying different tolerances to the different columns of a table of numbers.

Between lines, the user can place "embedded commands" in the input files. These commands are instructions to parser that can change what tolerances are attached to real numbers and the commenting and literal string conventions used by the parser. Embedded commands are flagged to the parser by starting the line with a user-specified escape string. By combining within line and between line differentiation, it is possible for the user to specify a different tolerance for every single real number in the input files.

---

[7] The problem of having the elements of tables become misaligned when the differencing algorithm is trying to find a minimal number of edits can be reduced somewhat by retaining newlines and not using tolerances. Unfortunately, it does not go away.

[8] A "change" can be expressed as one deletion and one insertion at the same point in the text.

[9] For instance, if one wants to have **spiff** produce output that can be fed into the **ed** editor.

Visual Mode

So far, **spiff's** operation as an intelligent filter has been described. **Spiff** also has an interactive mode. When operating in interactive mode, **spiff** places corresponding sections of the input files side by side on user's screen.[10] Tokens are compared using a one-to-one ordinal comparison, and any tokens that are found to be different are highlighted in reverse video. The user can interactively change the tolerances and **spiff** will alter the display to reflect which real numbers exceed the new tolerances. Other commands allow the user to page through the file and exit.

Performance

Two components of **spiff,** the parser and the differencing algorithm, account for most of the execution time. Miller and Myers compare their algorithm to the one used in the diff program. To restate their results, the Miller/Myers algorithm is faster for files that have relatively few differences but much slower (quadratic time) for files with a great many differences.

For cases where the files do not differ greatly, parsing the input files takes most of the time (around 80% of the total).[11] The performance of the parser is roughly similar to programs that do a similar level of parsing (i.e. programs that must examine each character in the file). For files where roughly half of the tokens are real numbers, **spiff** takes about twice as long to parse the input files as an **awk** program that counts the number of words in a file:[12]

```
awk '{total += NF}' firstfile secondfile
```

The time that it takes **spiff** to parse a file is substantially increased if scanning is done for comments and delimited literal strings. The precise effect depends upon the length of the delimiters, whether they are restricted to appear at beginning of line, and the frequency with which literals and comments appear in the input files. As an example, adding the 12 literal conventions[13] and 1 commenting convention required for C code roughly doubles the time required to parse input files.[14]

A more complete approach to evaluating **spiff's** performance must measure the total time that it takes for the user to complete a differencing task. For example, consider one of the test suites for the S statistical analysis package mentioned at the beginning of this paper. The output file for each machine is 427 lines long and contains 1090 floating point numbers. It takes **diff** approximately 2 seconds on one of our "6 MIPS"[15] computers to compare the two files and produce an edit script that is 548 lines long containing 1003 "differences" in the floating point numbers. It takes the average tester 5 minutes to print out the edit script and roughly 2 hours to examine the output by hand to determine that the machines are, in fact, both giving nearly identical answers. The total time needed is 2 hours 5 minutes and 2 seconds.

In contrast, it takes **spiff** approximately 6 seconds on one of our "6 MIPS" computers to produce an output file that is 4 lines long.[16] It takes the average tester 30 seconds to examine **spiff's** output. The total for **spiff** is 36 seconds. Therefore for this case, **spiff** will get the job done roughly 208.88 times faster than **diff.**

---

[10] Although the current implementation of **spiff** runs in many environments, interactive mode works only under the MGR window manager.[5] Other graphics interfaces will probably be added over time.

[11] No effort has yet been made to make the parser run more quickly. A faster parser could no doubt be written by generating a special state machine.

[12] For **awk,** a word is any string separated by white space.

[13] One literal convention is for C literal strings. The rest enumerate multicharacter operators.

[14] So in total, it takes **spiff** about 4 times longer to parse a C program than it takes **awk** to count the number of words in the same file.

[15] We will not comment on the usefulness of "MIPS" as a measure of computing speed. The numbers provided are only intended to give the reader some vague idea of how fast these programs run.

[16] The output would be zero length except that the output of the **time** command is built into the S tests. The timing information could easily be ignored using **spiff's** embedded commands. But, as we shall see, it hardly seems worth the trouble.

In general, it is misleading to compare **spiff's** speed with that of **diff.** While both programs are looking for differences between files, they operate on very different types of data (tokens vs. bytes). An analogous comparison could be made between the speed of an assembler and the speed of a C compiler. They are both language translators. One runs much faster than the other. None the less, most programmers use the slower program whenever possible.

Using Spiff For Making Regression Tests Of Software

We envision **spiff** to be the first of several tools for aiding in the now arduous task of making regression tests.[17] Given **spiff's** current capabilities, the regression test designer can take the output of an older version of software and through the use of literal string and commenting conventions, specify what parts of the output must remain identical and what sections can change completely. By specifying tolerances, the test designer can take into account how much of a difference in floating point calculations is acceptable.

The test designer is also free to edit the output from the older version of the software and add embedded commands that can instruct **spiff** to treat various parts of the output differently. The newly edited output can then serve as a template for the output of later versions of the software.

Obviously, editing output by hand is a very low level mechanism for adding specification information. It is our intention that **spiff** will become the last element in a pipeline of programs. Programs (as yet unwritten) located earlier in the pipeline can implement a higher level representation of the specification information. They read in the old and new input files, add the appropriate embedded commands, and then pass the results to **spiff** which will do the actual differencing.

Future Work

There are many features that could be added to **spiff** (if there are not too many already). Some of these include:

- Using separate differencing algorithms on separate sections of the file and/or limiting the scope of an edit sequence (fencing)
- Providing a more general mechanism for specifying comments and literals (perhaps allowing specification in terms of regular expressions). As yet, we have not encountered any important cases where regular expressions have been needed. Until such a case is encountered, we will leave regular expressions out in the name of simplicity.
- Allowing for a more general specification of what lines should look like. At present, the user can only specify tolerances for numbers as a function of their ordinal position on a line. The difficulty in expanding the specification abilities of **spiff** is knowing when to stop. In the extreme, we might add all of the functionality of a program such as **awk.**[18] We hope to keep **spiff** as simple as possible. Our first efforts in this direction will try to implement higher level specification functions outside of **spiff.**

Acknowledgements

First and foremost, we thank Stu Feldman for his endless patience, constant encouragement and numerous good ideas. We also extend thanks to Doug McIlroy for bringing the Miller/Myers algorithm to our attention, Nat Howard for a key insight and for his editorial comments and Steve Uhler and Mike Bianchi for their editorial comments.

---

[17] In software engineering parlance, a "regression test" is the process by which a tester checks to make sure that the new version of a piece of software still performs the same way as the older versions on overlapping tasks.

[18] Imagine handling the case such as "apply this tolerance to all numbers that appear on a line starting with the word 'foo' but only if the number is between 1.9 and 3.6 and the word 'bar' does not appear on the line".

References

[1] Hunt,J.W. and M.D. McIlroy. *An Algorithm For Differential File Comparisons,* **Bell Labs Computer Science Technical Report,** Number 41, 1975.

[2] Becker,R.A. and J.M. Chambers (1984). **S – An Interactive Environment For Data Analysis And Graphics.** Belmont, CA: Wadsworth Inc.

[3] Watterson, B. (1987). **Calvin and Hobbes.** New York: Andrews, McMeel & Parker.

[4] Miller, W. and E.W. Myers. *A File Comparison Program,* **Software – Practice and Experience** 15, 11, 1025-1040, 1985.

[5] Uhler, S.A. *MGR -- A Window Manager For UNIX,* Sun User's Group Meeting. September 1986.