2 Example: Finding File Differences

Program file for this chapter: `diff`

As an example of a practical program that manipulates data files, this chapter is about comparing two similar files to find the differences between them. This program is most often useful in comparing a current version of a file with an earlier version to see what has changed. On the next page is an example of two input files and the program's output. The output shows only those lines that differ between the two files, indicating changed lines, insertions, and deletions. When several consecutive lines are different, they are grouped together as a single reported change. (To demonstrate all of the program's capabilities, I've used short input files with more lines different than identical, and so the program's report is longer than the input files themselves. In a more realistic example, the input files might be hundreds of lines long, with only a few differences, so it would be easier to read the program's output than to examine the files directly.)

I've called this program `diff` because it was inspired by a similar program of that name in the Unix operating system. The format of the report that my `diff` generates is similar to that of the Unix version. In particular, I've followed the Unix `diff` convention that when a line from one of the input files appears in the report, it is marked by either a "<" character if it's from the first file or a ">" character if it's from the second file.

The numbers in the lines that begin with `CHANGE`, `INSERT`, or `DELETE` are line numbers, counting from one, in each of the two files. For example, the line

```
CHANGE 6-8 6-7
```

indicates that lines 6 through 8 in the first file were replaced by lines 6 through 7 in the second file. (The program considers a change to be finished when it finds two consecutive identical lines in the two files. In this case, lines 9 and 10 of the first file are identical to lines 8 and 9 of the second file.)

Input Files

Text1

```
My goal in this series of books
is to make the goals and methods
of a serious computer scientist
accessible, at an introductory
level, to people who are
interested in computer
programming but are not computer
science majors.
If you're an
adult or teenaged hobbyist,
or a teacher who wants to use the
computer as an educational tool,
you're definitely part of this
audience.
```

Text2

```
My goal in this series of books
is to make the goals and methods
of a mad computer scientist
accessible, at an introductory
level, to people who are
interested in playing computer
games.
If you're an
adult or teenaged hobbyist,
you're definitely part of this
audience.
And I hope you appreciate
the privilege!
```

Output File

```
DIFF results:
< File 1 = Text1
> File 2 = Text2
==========
CHANGE 3-3 3-3
< of a serious computer scientist
-----
> of a mad computer scientist
==========
CHANGE 6-8 6-7
< interested in computer
< programming but are not computer
< science majors.
-----
> interested in playing computer
> games.
==========
DELETE 11-12 10
< or a teacher who wants to use the
< computer as an educational tool,
==========
INSERT 15 12-13
> and I hope you appreciate
> the privilege!
==========
```

The `diff` procedure takes three inputs. The first two are names of the two input files; the third is either a name for an output file or an empty list, in which case the program's results are printed on the screen. For example, to see the results of my sample run, I'd say

```
diff "Text1 "Text2 []
```

I picked this project partly because it requires switching between two input files, so you can see how the program uses `setread` repeatedly.

Program Overview

`Diff` reads lines from the two input files in alternation. As long as the corresponding lines are equal, the program just moves on to the next pair of lines. (Procedure `diff.same` handles this process.) When a difference is found, the program's operation becomes more complicated. It must remember all the lines that it reads from both files until it finds two consecutive equal pairs of lines. (Procedures `diff.differ` and `diff.found` do this.)

Life would be simple if the differences between the two files were only changes within a line, without adding or removing entire lines. (This would be a realistic assumption if, for example, the second file had been created by applying a spelling correction program to the first file. Individual words would then be different, but each line of the second file would correspond to one line of the first.) In that case, the structure of `diff.differ` could be similar to that of `diff.same`: Read a line from each file, compare the two, and report the pairs that are different.

But in practice, a change to a paragraph may make the file longer or shorter. It may turn out, as in my sample run, that three lines from the first file correspond to only two lines from the second one. If that's the case, then there's no guarantee that the equal lines that mark the end of a change will be at the same line *numbers* in the two files. (In the sample, line 9 of the first file matches line 8 of the second.) Whenever the program reads a line from one file, therefore, it must compare that line to *every* line that it's read from the other file since the two started being different. Therefore, `diff.differ` must *remember* all of the lines that it reads from both files.

Finally, when two pairs of equal lines are found, the program must report the difference that it's detected. That's the job of procedure `report`. Once the change has been reported, the program continues in `diff.same` until another difference is found.

The program finishes its work when the ends of both input files have been reached.

The File Information Block Abstract Data Type

For each of the two input files, the program must remember several kinds of information. The `report` procedure must know which is file number 1 and which file number 2, in order to print the lines with the correct starting character. The name of each file is needed as the input to `setread`. The current line number is needed in order to report the location within each file of a changed section. As I've just explained, there is a collection of *pending* lines during the examination of a change; we'll see shortly that another collection of *saved* lines is used for another purpose.

To keep all the information for a file collected together, `diff` uses an abstract data type called a "file information block," or FIB, that is implemented as an array with five members. The array is made by a constructor procedure `makefile`, and there are selectors for four of the five components: `which`, `filename`, `linenum`, and `lines`. For the fifth component, the saved lines, instead of a selector for the entire collection the program uses a selector `popsaved` that outputs a single line each time it's invoked. (This will make more sense when you read about saved lines in the next section.)

The procedures within this program use these two FIBs as inputs instead of just the filenames. To read from one of the files, for example, the program will say

```
setread filename :fib1
```

Saving and Re-Reading Input Lines

One further detail complicates the program. Suppose that a change is found in which the two groups of differing lines are of different lengths. For example, suppose three lines in the first file turn into six lines in the second file, like this:

```
Original line 1                     Original line 1
Original line 2                     Changed line 2
Original line 3                     Changed line 3
Original line 4                     New line 3.1
Original line 5                     New line 3.2
Original line 6                     New line 3.3
Original line 7                     Changed line 4
Original line 8                     Original line 5
Original line 9                     Original line 6
```

The program has been reading lines alternately from the two files. It has just read the line saying "Original line 6" from the second file, and that's the second consecutive match with a line previously read from the first file. So the program is ready to report a change from lines 2–4 of the first file to lines 2–7 of the second.

The trouble is that the program has already read three lines of the first file (the last three lines shown above) that have to be compared to lines that haven't yet been read from the second file. Suppose that the files continue as follows:

```
Original line 10                  Original line 7
```

We can't just say, "Okay, we've found the end of a difference, so now we can go back to `diff.same` and read lines from the two files." If we did that, we'd read "Original line 10" from file 1, but "Original line 7" from file 2, and we'd think there is a difference when really the two files are in agreement.

To solve this problem we must arrange for `diff.same` to *re-read* the three unused lines from file 1. Logo allows a programmer to re-read part of a file by changing the current *position* within the file (this ability is called *random access*), but in this program I found it easier to *buffer* the lines by saving them in a list and then, the next time the program wants to read a line from the file, using one of the saved lines instead.

```
to readline :fib
if savedp :fib [output popsaved :fib]
setread filename :fib
output readword
end
```

The first instruction of this procedure says, "If there are any saved lines for this file, remove the first one from the list and output it." Otherwise, if there are no saved lines, then the procedure directs the Logo reader to the desired file (using `setread`) and uses `readword` to read a line. Because `popsaved` removes a line from the list of saved lines, eventually the saved lines will be used up and then the program will continue reading from the actual file.

Skipping Equal Lines

Here is the procedure that skips over identical pairs of lines:

```
to diff.same :fib1 :fib2
local [line1 line2]
do.while [make "line1 getline :fib1
          make "line2 getline :fib2
          if and listp :line1 listp :line2 [stop]    ; Both files ended.
] [equalp :line1 :line2]
addline :fib1 :line1                                 ; Difference found.
addline :fib2 :line2
diff.differ :fib1 :fib2
end
```

```
to getline :fib
nextlinenum :fib
output readline :fib
end
```

Most of the names you don't recognize are selectors and *mutators* for the FIB abstract data type. (A mutator is a procedure that changes the value of an existing datum, such as `setitem` for arrays.) One new Berkeley Logo primitive used here is `do.while`. It takes two inputs, an instruction list and an expression whose value is `true` or `false`. `Do.while` first carries out the instructions in the first input. Then it evaluates the predicate expression. If it's `true`, then `do.while` repeats the process, carrying out the instructions and evaluating the predicate, until the predicate becomes `false`. In this case, the idea is "Keep reading lines as long as `:line1` and `:line2` are equal."

`Getline` reads a line, either from the file or from the saved lines, and also adds one to the current line number by invoking `nextlinenum`:

```
to nextlinenum :fib
setitem 3 :fib (item 3 :fib)+1
end
```

This is a typical mutator; I won't show the others until the complete program listing at the end of the chapter.

Comparing and Remembering Unequal Lines

```
to diff.differ :fib1 :fib2
local "line
make "line readline :fib1
addline :fib1 :line
ifelse memberp :line lines :fib2 ~
       [diff.found :fib1 :fib2] ~
       [diff.differ :fib2 :fib1]
end
```

`Diff.differ` reads a line (perhaps a saved line) from one of the files, adds it to the collection of pending lines (not saved lines!) for that file, then looks to see whether a line equal to this one is pending in the other file. If so, then we may have found the end of the changed area, and `diff.found` is called to make sure there is a second pair of equal lines following these two. If not, we must read a line from the other file; this is accomplished by a recursive call to `diff.differ` with the two inputs in reversed order.

What was `:fib1` this time will be `:fib2` in the recursive call, and vice versa. (This is why the FIB data type must include a record of which is the original file 1 and file 2.)

The reason that `diff.differ` uses `readline` rather than `getline` to read from the input files is that it doesn't advance the line number. When dealing with a difference between the files, we are keeping a range of lines from each file, not just a single line. The line number that the program keeps in the FIB is that of the *first* different line; the line number of the last different line will be computed by the `report` procedure later.

```
to diff.found :fib1 :fib2
local "lines
make "lines member2 (last butlast lines :fib1) ~
                    (last lines :fib1) ~
                    (lines :fib2)
ifelse emptyp :lines ~
       [diff.differ :fib2 :fib1] ~
       [report :fib1 :fib2 (butlast butlast lines :fib1)
               (firstn (lines :fib2) (count lines :fib2)-(count :lines))]
end
```

`Diff.found` is called when the last line read from file 1 matches some line pending from file 2. Its job is to find out whether the last *two* lines from file 1 match two consecutive lines from file 2. Most of the work is done by the straightforward helper procedure `member2`, which works this way:

```
> show member2 "and "joy [she's my pride and joy etcetera]
[and joy etcetera]

> show member2 "pride "joy [she's my pride and joy etcetera]
[]
```

If the first two inputs are consecutive members of the third, then `member2` outputs the portion of its third input starting from the point at which the first input was found. If not, then `member2` outputs the empty list.

If `member2`'s output is empty, we continue reading lines from the two files by invoking `diff.differ`. If not, then we've found the end of a change, and we invoke `report` to print the results. The first two inputs to `report` are the two files; the third and fourth are the corresponding sets of unequal lines. The unequal lines from file 1 are all but the last two, the ones we just matched; the unequal lines from file 2 are all but the ones that `member2` output. Helper procedure `firstn` is used to select those lines.

Reporting a Difference

The `report` procedure is somewhat lengthy, but mostly because differences in which one of the sets of lines is empty are reported specially (as an insertion or a deletion, rather than as a change).

```
to report :fib1 :fib2 :lines1 :lines2
local [end1 end2 dashes]
if equalp (which :fib1) 2 [report :fib2 :fib1 :lines2 :lines1 stop]
print "==========
make "end1 (linenum :fib1)+(count :lines1)-1
make "end2 (linenum :fib2)+(count :lines2)-1
make "dashes "false
ifelse :end1 < (linenum :fib1) [
    print (sentence "INSERT :end1+1 (word (linenum :fib2) "- :end2))
] [ifelse :end2 < (linenum :fib2) [
    print (sentence "DELETE (word (linenum :fib1) "- :end1) :end2+1)
] [
    print (sentence "CHANGE (word (linenum :fib1) "- :end1)
                            (word (linenum :fib2) "- :end2))
    make "dashes "true
]]
process :fib1 "|< | :lines1 :end1
if :dashes [print "-----]
process :fib2 "|> | :lines2 :end2
diff.same :fib1 :fib2
end

to process :fib :prompt :lines :end
foreach :lines [type :prompt print ?]
savelines :fib butfirst butfirst chop :lines (lines :fib)
setlines :fib []
setlinenum :fib :end+2
end
```

Here's how to read `report`: The first step is to ensure that the files are in the proper order, so that `:fib1` is file number 1. (If not, `report` invokes itself with its inputs reordered.) The next step is to compute the ending line number for each changed section; it's the starting line number (found in the file data structure) plus the number of unmatched lines, minus one. `Report` then prints a header, choosing `INSERT`, `DELETE`, or `CHANGE` as appropriate. Finally, it invokes `process` once for each file.

`Process` prints the unmatched lines, with the appropriate file indicator (`<` or `>`). Then it takes whatever pending lines were not included in the unmatched group and transfers them to the saved lines, so that they will be read again. (As a slight efficiency improvement, `process` skips over the two lines that we know matched two lines in the other file; there's no need to read those again.) The set of pending lines is made empty, since no file difference is pending. Finally, the line number in the file structure is increased to match the position following the two lines that ended the difference.

If `process` confuses you, look back at the example I gave earlier, when I first talked about saving and re-reading lines. In that example, the lines from "Original line 7" to "Original line 9" in the first file are the ones that must be moved from the list of pending lines to the list of saved lines. (No lines will be moved in the second file, since that one had the longer set of lines in this difference, six lines instead of three.)

By the way, in the places where the program adds or subtracts one or two in a line number calculation, I didn't work those out in advance. I wrote the program without them, looked at the wrong results, and then figured out how to correct them!

Program Listing

I've discussed the most important parts of this program, but not all of the helper procedures. If you want to understand the program fully, you can read this complete listing:

```
to diff :file1 :file2 :output
local "caseignoredp
make "caseignoredp "false
openread :file1
openread :file2
if not emptyp :output [openwrite :output]
setwrite :output
print [DIFF results:]
print sentence [< File 1 =] :file1
print sentence [> File 2 =] :file2
diff.same (makefile 1 :file1) (makefile 2 :file2)
print "==========
setread []
setwrite []
close :file1
close :file2
if not emptyp :output [close :output]
end
```

```
;; Skip over identical lines in the two files.

to diff.same :fib1 :fib2
local [line1 line2]
do.while [make "line1 getline :fib1
          make "line2 getline :fib2
          if and listp :line1 listp :line2 [stop]    ; Both files ended.
] [equalp :line1 :line2]
addline :fib1 :line1                                 ; Difference found.
addline :fib2 :line2
diff.differ :fib1 :fib2
end

;; Remember differing lines while looking for ones that match.

to diff.differ :fib1 :fib2
local "line
make "line readline :fib1
addline :fib1 :line
ifelse memberp :line lines :fib2 ~
       [diff.found :fib1 :fib2] ~
       [diff.differ :fib2 :fib1]
end

to diff.found :fib1 :fib2
local "lines
make "lines member2 (last butlast lines :fib1) ~
                    (last lines :fib1) ~
                    (lines :fib2)
ifelse emptyp :lines ~
       [diff.differ :fib2 :fib1] ~
       [report :fib1 :fib2 (butlast butlast lines :fib1)
             (firstn (lines :fib2) (count lines :fib2)-(count :lines))]
end

to member2 :line1 :line2 :lines
if emptyp butfirst :lines [output []]
if and equalp :line1 first :lines equalp :line2 first butfirst :lines ~
   [output :lines]
output member2 :line1 :line2 butfirst :lines
end
```

```
to firstn :stuff :number
if :number = 0 [output []]
output fput (first :stuff) (firstn butfirst :stuff :number-1)
end

;; Read from file or from saved lines.

to getline :fib
nextlinenum :fib
output readline :fib
end

to readline :fib
if savedp :fib [output popsaved :fib]
setread filename :fib
output readword
end

;; Matching lines found, now report the differences.

to report :fib1 :fib2 :lines1 :lines2
local [end1 end2 dashes]
if equalp (which :fib1) 2 [report :fib2 :fib1 :lines2 :lines1 stop]
print "==========
make "end1 (linenum :fib1)+(count :lines1)-1
make "end2 (linenum :fib2)+(count :lines2)-1
make "dashes "false
ifelse :end1 < (linenum :fib1) [
    print (sentence "INSERT :end1+1 (word (linenum :fib2) "- :end2))
] [ifelse :end2 < (linenum :fib2) [
    print (sentence "DELETE (word (linenum :fib1) "- :end1) :end2+1)
] [
    print (sentence "CHANGE (word (linenum :fib1) "- :end1)
                            (word (linenum :fib2) "- :end2))
    make "dashes "true
]]
process :fib1 "|< | :lines1 :end1
if :dashes [print "-----]
process :fib2 "|> | :lines2 :end2
diff.same :fib1 :fib2
end
```

```
to process :fib :prompt :lines :end
foreach :lines [type :prompt print ?]
savelines :fib butfirst butfirst chop :lines (lines :fib)
setlines :fib []
setlinenum :fib :end+2
end

to chop :counter :stuff
if emptyp :counter [output :stuff]
output chop butfirst :counter butfirst :stuff
end

;; Constructor, selectors, and mutators for File Information Block (FIB)
;; Five elements: file number, file name, line number,
;; differing lines, and saved lines for re-reading.

to makefile :number :name
local "file
make "file array 5               ; Items 4 and 5 will be empty lists.
setitem 1 :file :number
setitem 2 :file :name
setitem 3 :file 0
output :file
end

to which :fib
output item 1 :fib
end

to filename :fib
output item 2 :fib
end

to linenum :fib
output item 3 :fib
end

to nextlinenum :fib
setitem 3 :fib (item 3 :fib)+1
end

to setlinenum :fib :value
setitem 3 :fib :value
end
```

```
to addline :fib :line
setitem 4 :fib (lput :line item 4 :fib)
end

to setlines :fib :value
setitem 4 :fib :value
end

to lines :fib
output item 4 :fib
end

to savelines :fib :value
setitem 5 :fib (sentence :value item 5 :fib)
end

to savedp :fib
output not emptyp item 5 :fib
end

to popsaved :fib
local "result
make "result first item 5 :fib
setitem 5 :fib (butfirst item 5 :fib)
output :result
end
```