A Tale of Two Tabulators

Richard A. O'Keefe
Computer Science, University of Otago

November 2001

Abstract

For a software engineering class, I present an example where a simple "obvious" English requirement led to two very different programs. This illustrates the need for getting the requirements from the client rather than inventing them yourself.

1 Background

Nathan Rountree (henceforth NR) and I (henceforth OK) have taught COSC 345, Software Engineering, for the last three years. Recently, NR was analysing some experimental results, and decided that it would be useful to have a program which would take output from a program and display it as an ASCII table. Here is an example of the kind of output one might get:

```
+-----------+----------------+
| gnats     | gram    $13.65 |
|           | each       .01 |
+-----------+----------------+
| gnu       | stuffed  92.50 |
| emu       |          33.33 |
+-----------+----------------+
| armadillo | frozen    8.99 |
+-----------+----------------+
```

NR mentioned to OK that he had written a program to do this in about 30 lines of Python. OK thought that sounded pretty neat, and while waiting for a copy of NR's program, decided to write his own. OK's final version (in appendix B) is 83 SLOC of AWK, NR's final version (in appendix A) is 15 SLOC of Python.

If you have two programs for the same problem, and one of them is about five and a half times smaller than the other, then either one of the programmers is much better than the other or one of the languages is much better than the other.

NR and OK agree that Python is a much better language than AWK. In this case its lists and higher-order functions helped. AWK doesn't even have a built-in `max` function.

Far more interesting from a Software Engineering viewpoint, however, is that they *aren't* solving the same problem. The reason for this, and the consequences of the difference, are illuminating.

When OK wrote an AWK version of NR's program (in appendix C), it was 20 SLOC, which does not suggest a huge advantage for NR or Python in this case. It really is the problem that is different. We can see this by comparing how much code each program devotes to each task.

nrtable.py	nrtable.awk	oktable.awk	Task
N/A	N/A	22	parse format
5	8	11	load data
N/A	N/A	13	non-“r” alignment
5	6	17	print borders
4	5	6	other printing
N/A	N/A	11	error checking

2 The common requirements

The common requirement was quite informal: “write a program to read line-oriented data in a reasonable way and write out a table”.

NR and OK both understood the requirements to include

1. Each data line turns into one output line, in the given order.
2. Each output line is either a transformed data line, or a border line.
3. Table cells are aligned by adding spaces. Tab characters should not be used.
4. Table and cell borders may be drawn using `|`, `-`, and `+` at crossings.
5. There should be at least one blank between the text in a cell and the next visible character.
6. Each column should be as narrow as possible, subject to all its cells fitting.
7. Each output line should be as narrow as possible, subject to all its cells fitting. Trailing blanks do not count.
8. Leading and trailing white space characters in a cell are to be stripped off and do not count towards a column width.
9. One possible form of input has fields separated by commas.
10. One possible method of alignment is right alignment, where the last visible character of a cell is as far to the right as possible.
11. Only modest amounts of data are to be displayed, so the speed of the program is not particularly important.
12. The program should not have any fixed limits.
13. Fields may be assumed to be of reasonable length so that line breaking will never be required.
14. Input with no columns need not be possible; input with one column must be possible and work sensibly. Input with no rows must be possible and work sensibly.

That’s an impressive list of detailed requirements to agree on, simply by sharing a common “culture”.

3 The causes of divergence

NR started with special-purpose programs, and decided to simplify them by having them generate unaligned data and writing a single table-aligning filter. It seemed reasonable to him to modify the output of the data-generating programs to suit the filter. His data were whole numbers.

NR’s requirements therefore included:

15. The input format may be chosen to suit the convenience of the filter.

16. Only comma separation need be supported.

17. Only right alignment need be provided.

18. Individual control of cell borders is needed.

19. There is no need to check the input for plausibility.

20. All input lines have the same number of fields, so it is acceptable to use the minimum field count for all lines.

21. It is acceptable to reserve "`|`", "`+`", and "`-`" as fields.

OK started with existing data sets and data-generating programs which he was unwilling (in some cases) or unable (in others) to modify. None of his existing data or programs used commas as separators; all used the more UNIX-traditional white space or TAB separators. OK's data included labels, whole numbers, and decimal fractions.

OK's requirements therefore included:

15. The format of existing data and data sources must be respected.

16. Any input field separation that traditional UNIX tools support should be supported; the default field separation should be the default for AWK.

17. Left alignment (for labels), right alignment (for whole numbers), and decimal point alignment (for decimal fractions) must be supported.

18. It must be possible to provide a reasonable table including borders (but not necessarily internal rules) without touching the data lines at all. This leads to the conclusion that individual control of cell borders is impossible, because there is no place to put the information.

19. The format needs to be checked and a warning should be issued if the output lines are too wide for the screen. (OK is fed up with students handing in program listings with large chunks of code disappearing off the right hand side of the page, and is really annoyed with Web pages that lose a word or two off the right margin when printed, including some of the Python material.)

20. Input lines may have varying numbers of fields. Losing data is unacceptable, so the maximum field count must be used for all lines.

21. It is acceptable to reserve "`-`" and "`=`" as entire lines, but it is not acceptable to reserve any special fields in multicolumn tables.

What determined the choice of programming language? Habit. OK agrees that Python is the better language, but knows AWK better, and it was up to the job. For a 6000-line test case, Python took 19.1 seconds, Nawk took 16.1 seconds, Gawk took 10.7 seconds, and Mawk took 6.0 seconds. For realistic inputs, speed really is *not* an issue for this problem.

4 Divergent input formats

The input to NR's program has the form

input $\longrightarrow$ line*
line $\longrightarrow$ field {'`,`' field}* eol
field $\longrightarrow$ '`|`' | '`+`' | '`-`' | text

The '`|`' and '`+`' fields are copied literally to the output; they are used for vertical borders. The '`-`' character is replicated to the width of the cell, plus 2. Other fields are padded on the left to the width of the cell, and have an extra blank added at each end.

Here is the input that produces the table in section 1:

```
+,-,+,-,-,+
|,gnats,|,gram,$13.65,|
|,,|,each,.01|
+,-,+,-,-,+
|,gnu,|,stuffed,92.50,|
|,emu,|,,33.33,|
+,-,+,-,-,+
|,armadillo,|,frozen,8.99,|
+,-,+,-,-,+
```

The input to OK's program has the form

input $\longrightarrow$ {format eol}$^?$ line*
line $\longrightarrow$ {'`-`' | '`=`' | fields} eol
fields $\longrightarrow$ text {FS text}*

format $\longrightarrow$ {'`-`' | '`=`'}$^?$ '`|`'$^?$ {{'`l`' | '`r`' | '`c`' | '`d`'} '`|`'$^?$}* {'`-`' | '`=`'}$^?$

where field separation is done using AWK rules.

The '`-`' and '`=`' characters stand for single and double horizontal rules. If the format begins (ends) with such a character, the output begins (ends) with such a rule. The '`|`' character stands for a vertical rule at the beginning of a line or after a cell. The letters `l`, `r`, `c`, and `d` stand for left, right, centred, and decimal point alignment respectively.

The format may be given on the command line using `-v format=`*format*; if it is not, the first input line will be taken as the format. With the exception of horizontal rules properly inside a table, all formatting can be done by providing a format command line argument without altering existing data.

Here is the input that produces the table in section 1:

```
-|l|lr|-
gnats,gram,$13.65
,each,.01
-
gnu,stuffed,92.50
emu,,33.33
-
armadillo,frozen,8.99
```

The only inputs that are acceptable to both programs have

1. comma-separated fields with
2. no borders and
3. only right alignment.

5 The consequences of divergence

If NR were the customer, he would reject OK's program because it does not provide individual control of cell borders. It would be less work for OK to write a new program than to "fix" the existing one, and it cannot be done without destroying the properties OK wants.

If OK were the customer, he would reject NR's program because it cannot work with space-separated fields, cannot do left alignment, and cannot do decimal point alighment. It would be less work for NR to write a new program than to "fix" the existing one, and it cannot be done without destroying the properties NR wants.

Reconciling the two programs is not possible without negotiating new and more specific requirements.

It is worth noting that *neither* program can handle "real" CSV data produced by typical spread-sheet programs. Given input like

```
"Doe, John", bass, 2000-08-10
"Roe, Richard", tenor, 2001-02-12
```

both programs would produce output resembling

```
"Doe     John"  bass 2000-08-10
"Roe  Richard" tenor 2001-02-12
```

instead of the correct output

```
   Doe, John  bass 2000-08-10
Roe, Richard tenor 2001-02-12
```

Neither OK nor NR was concerned with data produced by a spread-sheet, although other users might well be.

The Moral

The moral of the tale is that even for a problem this simple, it is a mistake to make up your own answers to questions about requirements. What seems most obvious and most useful to you may be neither obvious nor useful to the person who wants the program.

Most dangerous of all are the gaps we are not *conscious* of filling. There are bound to be points among the 14 common requirements that other potential users would disagree with.

At 21 identified requirements, NR's program has more requirements than lines of code. Even OK's program has one requirement for every 4 SLOC. The higher the level of the programming language you use, the higher you can expect this ratio to be, although not perhaps this extreme. The comments were stripped out of the programs; both NR's real code and OK's real code have more comment lines than SLOC.

Appendix A: NR's program

The `nrtable.py` script reads comma-separated data from stdin and writes a neatly formatted table to stdout, assuming fixed-width characters. It was written by Nathan Rountree in November 2001. White space characters are ignored at the beginning and end of fields.

```
#!/usr/bin/env python
import sys, string
rows = map(lambda x: map(string.strip, string.split(x, ",")),
           string.split(string.strip(sys.stdin.read()), "\n"))
col_widths = map(max, apply(map, (None,) +
                        tuple(map(lambda r: map(len, r), rows))))
for row in rows:
    for item in map(None, col_widths, row):
        if item[1] == '-':
            sys.stdout.write("-" * item[0] + "--")
        elif item[1] == '+' or item[1] == '|':
            sys.stdout.write("%s" % item[1])
        else:
            sys.stdout.write(" %*s " % item[:2])
    sys.stdout.write("\n")
```

Appendix B: OK's program

The `oktable.awk` script reads data from stdin with fields separated in the usual AWK fashion, and writes a neatly formatted table to stdout, assuming fixed-width characters. It needs a format line which may be provided on the command line as `-v format=`*format*, or may be the first data line. It was written by Richard O'Keefe in November 2001. White space characters are ignored at the beginning and end of fields. The first draft used user-defined functions; this version has been rewritten so that it can be run by the old `awk` interpreter as well as `nawk`, `mawk`, and `gawk`.

```
#!/bin/nawk -f
NR == 1 {
    x = tolower(s = format == "" ? $0 : format)
    gsub(/[ \t]+/, "",  x)
    if (match(x, /^[-=]/)) {
        divider[0] = substr(x, RSTART, RLENGTH)
        x = substr(x, RSTART+RLENGTH)
    }
    if (match(x, /[-=]$/)) {
        divider[-1] = substr(x, RSTART, RLENGTH)
        x = substr(x, 1, RSTART-1)
    }
    if (x ~ /[^|lcrd]/) {
        print "Bad header line:", s >>"/dev/stderr"
        exit 1
    }
    a = "l"
    if (bar[0] = x ~ /^[|]/) x = substr(x, 2)
    for (ncols = 1; x != ""; ncols++) {
        if (x !~ /^[|]/) {
            a = substr(x, 1, 1)
            x = substr(x, 2)
        }
        align[ncols] = a
        if (bar[ncols] = x ~ /^[|]/) x = substr(x, 2)
    }
    if (ncols > 1) ncols--
    if (format == "") next
}

/^[-=]$/ {
    divider[nrows] = $0
    next
}

{
    for (; ncols < NF; ncols++) {
        bar[ncols+1] = bar[ncols]
        align[ncols+1] = align[ncols]
    }
    nrows++
    for (i = 1; i <= NF; i++) {
        s = $i
        sub(/^[ \t]+/, "", s)
        sub(/[ \t]+$/, "", s)
```

```
        m = length(s)
        if (align[i] == "d") {
            n = index(s, ".")
            if (n != 0) {
                n = m - n + 1
                m = m - n
                if (n > after[i]) after[i] = n
            }
        }
        if (m > width[i]) width[i] = m
        item[nrows,i] = s
    }
}

END {
    if (divider[nrows] == "") divider[nrows] = divider[-1]
    for (i = 1; i <= ncols; i++) width[i] += after[i]
    n = -1
    if (bar[0]) n += 2
    for (i = 1; i <= ncols; i++) {
        n += width[i] + 1
        if (bar[i]) n += 2
    }
    if (n >= 80)
        print "Warning: output lines have", n,
              "characters." >>"/dev/stderr"
    nrows += 0
    for (r = 0; r <= nrows; r++) {
        if (r != 0) {
            if (bar[0]) printf "| "
            for (i = 1; i <= ncols; i++) {
                s = item[r, i]
                a = align[i]
                if (a == "l") {
                    printf "%-*s", width[i], s
                } else if (a == "r") {
                    printf "%*s", width[i], s
                } else if (a == "c") {
                    n = width[i] - length(s)
                    m = int(n/2)
                    printf "%*s%-*s", m, "", n-m, s
                } else {
                    n = width[i] - length(s)
                    p = index(s, ".")
                    m = p == 0 ? after[i] : after[i] - (length(s) - p + 1)
                    printf "%*s%s%*s", n-m, "", s, m, ""
                }
                if (bar[i]) printf " |"
                if (i < ncols) printf " "
            }
            printf "\n"
        }
        s = divider[r]
        if (s != "") {
```

```
            if (bar[0]) printf "+%s", s
            for (i = 1; i <= ncols; i++) {
                for (m = width[i]; m >= 1; m--) printf "%s", s
                if (bar[i]) printf "%s+", s
                if (i < ncols) printf "%s", s
            }
            printf "\n"
        }
    }
}
```

Appendix C: NR's program rewritten in AWK

The `nrtable.awk` script reads AWK-delimited data from stdin and writes a neatly formatted table to stdout, assuming fixed-width characters. It was written by Richard O'Keefe in November 2001. White space characters are ignored at the beginning and end of fields. If you want comma-separated fields, you must use the `-F,` (dash capital-F comma) command line option.

```
#!/bin/awk -f
{
    if (NF > ncols) ncols = NF
    nrows++
    for (c = 1; c <= NF; c++) {
        x = $c
        sub(/^[ \t]+/, "", x)
        sub(/[ \t]+$/, "", x)
        if (length(x) > width[c]) width[c] = length(x)
        field[nrows,c] = x
    }
}

END {
    for (r = 1; r <= nrows; r++) {
        for (c = 1; c <= ncols; c++) {
            x = field[r,c]
            if (x == "|" || x == "+") {
                printf x
            } else
            if (x == "-" || x == "=") {
                for (j = width[c]+2; j >= 1; j--) printf x
            } else {
                printf " %*s ", width[c], x
            }
        }
        printf "\n"
    }
}
```