Quick and Dirty Databases with Nice Output: AWK and TEX

Erich Neuwirth
Institute for Statistics and Computer Science, University of Vienna
Universitätsstraße 5-9, A-1010 Vienna, Austria
Bitnet: a4422dab@awiuni11

Abstract

This paper will describe an easy-to-use set of tools which enables moderately to well experienced TEX-users to produce formatted output from flat data files. Usually all you need is AWK, TEX, and a sort program.

A First Example

Let us assume we have a text file containing addresses in more or less random order. Let us further assume that the main purpose of the file is to produce a printed address list and directory, possible in different versions sorted according to different criteria. Let us further assume that the data file has a very simple structure, namely a number of lines for each address and different addresses being separated by blank lines. Additionally, we assume that the items to be used in sorting occur on the same line in every address block, e.g. the surname is always on the first line, the city on the third line, and so on.

Now we would like to format this material into a two-column format for an address book to be typeset by LATEX. For the first step, let us assume that the data already are sorted in the sequence we need for the printed version. Then the following little AWK program does the trick:

```
BEGIN   {
        RS=""
        FS="\n"
        printf "\\documentstyle[twocolumn]"
        printf "{article}\n"
        printf "\\begin{document}\n\n"
        }
        {
        for(i=1;i<=NF-1;i++)
            printf "%s  \\\\ \n",  $i
        printf "%s \n\n", $NF
        }

END     {
        print ""
        printf "\% %d records transferred \n\n"
            ,NR
        print "\\end{document}"
        }
```

If you keep the program in a file address.awk and execute the following command,

```
awk -f address.awk address.dat>address.tex
```

then the file address.tex will contain a LATEX file which prints the address book.

The command line will work with a UNIX or an MS-DOS system, if you have AWK. We will talk about the availability of AWK later.

Let us look at this program. It consists of three parts.

The first part, starting with BEGIN, is executed before any data is read from the input file. So, this is the place to put the LATEX preamble and any definitions you need for page layouts, (like page sizes, \parskip, etc.).

The middle part of our program, the one with the for loop, is executed for each record read in from the data file. This is where we can reformat the unformatted records from our primary data file.

The third part of the program, beginning with END, is executed after all input data was read, so this is the place for any final housekeeping activities and for \end{document}.

Here we are using the fact that AWK has an implicit loop running through all records in the data file. It also has easy-to-use tools for structuring the records into fields. "$i" in our program refers to the ith field of each record, so AWK by itself breaks the records into fields which are exactly the units we want to deal with for the TEX output.

In this program we use print and printf the same way it is used in C; the print command uses default printing formats; printf explicitly needs a formatting string.

The program by itself should be understandable to anybody with a little knowledge of C. The only additional knowledge is about the implicit loop over records.

If we want to inhibit page or column breaks for each entry in our address book, we simply add the AWK commands to make a "minipage" for each entry, so the middle part should look like this:

```
{
print "\\begin{minipage}{\\textwidth}"
for(i=1;i<=NF-1;i++)
    printf "%s  \\\\ \n",  $i
printf "%s \n", $NF
print "\\end{minipage}"
printf "\n"
}
```

The double backslashes are needed because in AWK, like in many C-like languages, the backslash is the escape character, so a double backslash is needed to produce a single backslash in the output file. If we want the first entry to be printed in boldface, only a slight change in our program is needed:

```
{
print "\\begin{minipage}{\\textwidth}"
printf "{\\bf %s }", $1
for(i=2;i<=NF-1;i++)
    printf "%s  \\\\ \n",  $i
printf "%s \n", $NF
print "\\end{minipage}"
printf "\n"
}
```

Of course this model can be generalized; we can extract any field from the record and add special formatting commands to the field. Using this little program as a skeleton, we already are able to reformat a simple text file with addresses to get a booklet with typeset quality.

Extending the Example

Besides the implicit loop, AWK has a strong pattern matcher directly built into the language. It uses the regular expression syntax of UNIX (like grep and other text-oriented tools in UNIX) and allows selection of records according to certain patterns. If, in our example, we wanted to print the address book for the inhabitants of Cork only, we would only have to change the middle part of the program very slightly:

```
/Cork/  {
        for(i=1;i<=NF-1;i++)
            printf "%s  \\\\ \n",  $i
        printf "%s \n\n", $NF
        }
```

So we only added the /Cork/ expression which tells AWK to perform the printing actions only to records containing the string Cork.

In this case, the command printing the number of records into the TEX file would print the number of records in the input file and not the number of addresses formatted. To correct this we have to modify our program again; we simply add a variable which counts the number or records transferred to the output file,

```
/Cork/  {
        nofout++
        for(i=1;i<=NF-1;i++)
            printf "%s  \\\\ \n",  $i
        printf "%s \n\n", $NF
        }
```

and we change the print command for NR in the end part of our program to:

```
printf "\% %d records transferred \n\n"
        ,nofout
```

AWK initializes variables with zero, so the counting works correctly. We will not explain the pattern-matching mechanism of AWK in this paper, that would go beyond both the scope and the space limits of the paper. We only wanted to demonstrate that selection of records is already built into AWK, so we do not need a special tool for that.

Sorting the Data

AWK by itself does not offer built-in capabilities for sorting data. It would be possible to write a sorting program in AWK, but since AWK is interpreted, performance would be low. It also would be a waste of energy for the programmer, since any reasonable operating system has a sort utility. The only task we have to perform is to write a small AWK program which reformats the input file in such a way that the sort program can sort it according to our criteria. We might have to write another small program which transforms the sorted data back into the original form. We need a little bit of additional information. In the example program in the beginning, we had the following code:

```
RS=""
FS="\n"
```

The variable RS defines the record separator; the variable FS defines the field separator. These two separators refer to the input file. The corresponding two variables, ORS and OFS, refer to the output file. Using these variables, we can transform multiline records into oneline records which can be better dealt with by sort programs. Consider the following program:

```
BEGIN   {
        RS=""
        FS="\n"
        ORS="\n"
        OFS="\t"
        }
```

```
{
printf "%10s  ", $3
print $1,$2,$3,$4,$5
}
```

The BEGIN part defines the record and field separators, the main loop writes the third variable to a fixed length field at the beginning of each output record and then writes all variables with the new output separator (a tab character) into the output file. So the output file contains one line per record. Now we simply use a sort program to sort the intermediate file into the order we need. Then we have to write a similar small AWK program which cuts off the first field of our new records and copies the remaining fields restoring the old multiline structure by using,

```
ORS="\n\n"
OFS="\n"
```

(ORS="" does *not* work). We now have the sorted data in the original format and can print the address book in the order we need.

General Considerations

The examples in the previous sections show that it is very easy to use AWK as a front end between data files and TeX. It is very easy to write short AWK programs which perform the simple formatting tasks needed for preparing data for TeX output. We even might state that we have a tool modeled after WEAVE, which is used in preparing TeX itself and which transforms "raw" input in the form of program parts and comments into a format which allows TeX to typeset itself and other programs with TeX typographical standard. In our case, we use AWK to transform "raw" data into TeX input.

We do not have the full power of database management systems, but if the data file only functions as a central repository of the data and the main final output is printed material, it is worthwhile to consider AWK as a solution.

Availability of AWK

AWK is part of UNIX in all its varieties, so it is widely available already.

The Free Software Foundation and its GNU project have produced a version called GAWK which is completely free and available on many file repositories on electronic networks. It also has been ported to many different operating systems, and it is available for MS-DOS.

The literature needed to handle projects like the ones described is: TeX related books, which the author is not going to cite just to be different from almost any other paper in these proceedings, and *The AWK Programming Language.*

Bibliography

Aho, A., Kernighan, B., and Weinberger, P. *The AWK Programming Language*, Addison-Wesley 1988.