The AWK Programming Language

Second Edition

Alfred V. Aho
Brian W. Kernighan
Peter J. Weinberger

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

The AWK Programming Language

Second Edition

algorithm uses recursion to handle negative numbers: if the input is negative, the function `addcomma` calls itself with the positive value, tacks on a leading minus sign, and returns the result. Note the `&` in the replacement text for `sub` to add a comma before each triplet of numbers.

Here are the results for some test data:

```
0                           0.00
-1                         -1.00
.1                          0.10
-12.34                    -12.34
-12.345                   -12.35
12345                  12,345.00
-1234567.89        -1,234,567.89
-123.                    -123.00
-123456              -123,456.00
```

Exercise 4-5. Modify `sumcomma`, the program that adds numbers with commas, to check that the commas in the numbers are properly positioned. □

Fixed-Field Input

Information appearing in fixed-width fields may require preprocessing before it can be used directly. Some programs print information in fixed columns, rather than with field separators; if the fields are too wide, the columns abut.

Fixed-field data is best handled with `substr`, which can be used to pick apart any combination of columns. For example, the output of the Unix `ls` command is formatted so everything lines up:

```
total 3024
drwxr-xr-x  9 bwk  staff      288 Mar  7  2019 Album Artwork
drwxr-xr-x  4 bwk  staff      128 Mar  7  2019 Previous iTunes Libraries
-rw-r--r--@ 1 bwk  staff    73728 Jul  3 19:34 iTunes Library Extras.itdb
-rw-r--r--@ 1 bwk  staff    32768 Jul 16  2016 iTunes Library Genius.itdb
-rw-r--r--@ 1 bwk  staff  1377841 Jul  3 19:34 iTunes Library.itl
drwxr-xr-x  6 bwk  staff      192 May 15  2020 iTunes Media
-rw-r--r--@ 1 bwk  staff        8 Jul  3 19:34 sentinel
```

The filenames are at the end but because they may contain spaces, they have to be extracted either with a function like `rest(9)`, which we described in Section 2.6, or with `substr`, as in this example:

```
{ print substr($0, index($0, $9)) }
total 3024
Album Artwork
Previous iTunes Libraries
iTunes Library Extras.itdb
iTunes Library Genius.itdb
iTunes Library.itl
iTunes Media
sentinel
```

Notice the use of `index` to compute what column the filename begins in.

Exercise 4-6. This code doesn't work if the filename in `$9` also appears as a substring earlier on the line. Fix it. □

Program Cross-Reference Checking

Awk is often used to extract information from the output of other programs. Sometimes that output is merely a set of homogeneous lines, in which case field-splitting or `substr` operations are sufficient. Sometimes, however, the upstream program thinks its output is intended for people. In that case, the task of the Awk program is to undo careful formatting, so as to extract the information from the irrelevant. The next example is a simple instance.

Large programs are built from many files. It is convenient (and sometimes vital) to know which file defines which function, and where the function is used. To that end, the Unix program `nm` prints a neatly formatted list of the names, definitions and addresses, and uses of the names in a set of object files. A typical fragment of its output looks like this:

```
lex.o:
0000000000000000 T startreg
                 U strcmp
00000000000003d0 T string
                 U strlen
                 U strtod

lib.o:
00000000000002f0 T eprint
00000000000015f0 T errcheck
0000000000000680 T error
                 U exit
                 U fclose
```

Lines with one field (e.g., `lex.o`) are filenames, lines with two fields (e.g., `U` and `fclose`) are uses of names, and lines with three fields are definitions of names. `T` indicates that a definition is a text symbol (function) and `U` indicates that the name is undefined.

Using this raw output to determine what file defines or uses a particular symbol can be a nuisance, because the filename is not attached to each symbol. For a C program the list can be long — it's 750 lines for the nine files of source that make up Awk itself. A three-line Awk program, however, can add the name (without the colon) to each item:

```
# nm.format - add filename to each nm output line

NF == 1 { sub(/:/,""); file = $1 }
NF == 2 { print file, $1, $2 }
NF == 3 { print file, $2, $3 }
```

The output from `nm.format` on the data shown above is

```
lex.o T startreg
lex.o U strcmp
lex.o T string
lex.o U strlen
lex.o U strtod
lib.o T eprint
lib.o T errcheck
lib.o T error
lib.o U exit
lib.o U fclose
```

Now it's easy for other programs to search this output or process it further.

This technique does not tell us where or how many times a name appears in a file, but these things can be found by a text editor or another Awk program. Nor does it depend on which language the programs are written in, so it is more flexible than the usual run of cross-referencing tools, and shorter and simpler too.

4.2 Data Validation

Another common use for Awk programs is data validation: making sure that data is legal or at least plausible. We saw some specific examples in Chapter 3 when we were looking at data from the Titanic. This section describes several small general-purpose programs that check input for validity. For example, consider the column-summing programs in the previous section. Are there any numeric fields where there should be nonnumeric ones, or vice versa? Such a program is close to one we saw before, with the summing removed:

```
# colcheck - check consistency of columns
#   input:  rows of numbers and strings
#   output: lines whose format differs from first line

NR == 1   {
    nfld = NF
    for (i = 1; i <= NF; i++)
       type[i] = isint($i)
}
{   if (NF != nfld)
       printf("line %d has %d fields instead of %d\n",
          NR, NF, nfld)
    for (i = 1; i <= NF; i++)
       if (isint($i) != type[i])
          printf("field %d in line %d differs from line 1\n",
             i, NR)
}

function isint(n) { return n ~ /^[+-]?[0-9]+$/ }
```

This certainly doesn't check for all possible errors. The test for integers is again just a sequence of digits with an optional sign; see the discussion of regular expressions in Section A.1.4 of the reference manual for a more complete explanation.

Balanced Delimiters

In the machine-readable text of this book, each program is introduced by a line beginning with `.P1` and is terminated by a line beginning with `.P2`. These lines are text-formatting commands that make the programs come out in their distinctive font when the text is typeset. Since programs cannot be nested, these text-formatting commands must form an alternating sequence

```
.P1 .P2 .P1 .P2 ... .P1 .P2
```

If one or the other of these delimiters is omitted, the output will be badly mangled by our text formatter. To make sure that the programs will be typeset properly, we wrote this tiny delimiter checker, which is typical of a large class of such programs:

```
# p12check - check input for alternating .P1/.P2 delimiters

/^\.P1/ { if (p != 0)
              print ".P1 after .P1 at line", NR
          p = 1
        }
/^\.P2/ { if (p != 1)
              print ".P2 with no preceding .P1 at line", NR
          p = 0
        }
END     { if (p != 0) print "missing .P2 at end" }
```

If the delimiters are in the right order, the variable p silently goes through the sequence of values 0 1 0 1 0 ... 1 0. Otherwise, the appropriate error messages are printed. We use a larger version to check the manuscript of the book for similar errors.

Exercise 4-7. What's a good way to extend this program to handle multiple sets of delimiter pairs or nested delimiters? □

Password-File Checking

The password file on a Unix system used to contain the names of and other information about authorized users. Each line of the password file had 7 fields, separated by colons:

```
root:qyxRi2uhuVjrg:0:2::/:
bwk:1L./v6iblzzNE:9:1:Brian Kernighan:/usr/bwk:
ava:otxs1oTVoyvMQ:15:1:Al Aho:/usr/ava:
uucp:xutIBs2hKtcls:48:1:uucp daemon:/usr/lib/uucp:uucico
pjw:xNqy//GDc8FFg:170:2:Peter Weinberger:/usr/pjw:
...
```

The first field is the user's login name, which should be alphanumeric. The second is an encrypted version of the password; if this field is empty, anyone can log in pretending to be that user, while if there is a password, only people who know the password can log in. The third and fourth fields are supposed to be numeric. The sixth field, the user login directory, should begin with /. The following program prints all lines that fail to satisfy these criteria, along with the number of the erroneous line and an appropriate diagnostic message.

```
# checkpasswd - check password file for correct format

BEGIN { FS = ":" }
NF != 7 {
    printf("line %d, does not have 7 fields: %s\n", NR, $0) }
$1 ~ /[^A-Za-z0-9]/ {
    printf("line %d, nonalphanumeric user id: %s\n", NR, $0) }
$2 == "" {
    printf("line %d, no password: %s\n", NR, $0) }
$3 ~ /[^0-9]/ {
    printf("line %d, nonnumeric user id: %s\n", NR, $0) }
$4 ~ /[^0-9]/ {
    printf("line %d, nonnumeric group id: %s\n", NR, $0) }
$6 !~ /^\// {
    printf("line %d, invalid login directory: %s\n", NR, $0) }
```

This is a good example of a program that can be developed incrementally: each time someone thinks of a new condition that should be checked, it can be added, so the program steadily becomes more thorough.

Generating Data-Validation Programs

We constructed the password-file checking program by hand, but a more interesting approach is to convert a set of conditions and messages into a checking program automatically. Here is a small set of error conditions and messages, where each condition is a pattern from the program above. The error message is to be printed for each input line where the condition is true.

```
NF != 7             does not have 7 fields
$2 == ""            no password
$1 ~ /[^A-Za-z0-9]/ nonalphanumeric user id
```

The following program converts these condition-message pairs into a checking program:

```
# checkgen - generate data-checking program
#     input:  expressions of the form: pattern tabs message
#     output: program to print message when pattern matches

BEGIN { FS = "\t+" }
{ printf("%s {\n\tprintf(\"line %%d, %s: %%s\\n\",NR,$0) }\n",
      $1, $2)
}
```

The output is a sequence of conditions and the actions to print the corresponding messages:

```
NF != 7 {
     printf("line %d, does not have 7 fields: %s\n",NR,$0) }
$2 == "" {
     printf("line %d, no password: %s\n",NR,$0) }
$1 ~ /[^A-Za-z0-9]/ {
     printf("line %d, nonalphanumeric user id: %s\n",NR,$0) }
```

When the resulting checking program is executed, each condition will be tested on each line, and if the condition is true, the line number, error message, and input line will be printed. Note that in `checkgen`, some of the special characters in the `printf` format string must be quoted to produce a valid generated program. For example, `%` is preserved by writing `%%` and `\n` is created by writing `\\n`.

This technique in which one Awk program creates another one is broadly applicable, and of course it's not restricted to Awk programs.

By the way, as a historical note, one of the inspirations for Awk was an error-checking tool created by Marc Rochkind at Bell Labs in the mid 1970s. Marc's program, written in C, took a sequence of regular expressions as input and created a C program that would scan its input and report any line that matched any of the patterns. It was a very neat idea, and we stole it unabashedly.

4.3 Bundle and Unbundle

Consider how to combine ("bundle") a set of text files into one file in such a way that they can be easily separated ("unbundled") into the original files. This section contains two tiny Awk programs that do this pair of operations. They can be used for bundling small files together to save disk space, or to package a collection of files for convenient emailing.

The `bundle` program is trivial, so short that you can just type it on a command line. All it does is prefix each line of the output with the name of the file, which comes from the built-in variable `FILENAME`.

```
# bundle - combine multiple files into one

{ print FILENAME, $0 }
```

The matching `unbundle` is only a little more elaborate:

```
# unbundle - unpack a bundle into separate files

$1 != prev { close(prev); prev = $1 }
           { print substr($0, index($0, " ") + 1) >$1 }
```

The first line of `unbundle` closes the previous file when a new one is encountered. If bundles don't contain many files (less than the limit on the number of simultaneously open files), closing the file isn't necessary.

By the way, the `>$1` in the last line of `unbundle` is not a relational operator, but causes the output to be written to a file whose name is stored in `$1`.

There are other ways to write `bundle` and `unbundle`, but the versions here are the easiest, and for short files, reasonably space efficient. Another organization is to add a distinctive line with the filename before each file, so the filename appears only once.

Exercise 4-8. Note that `bundle` assumes that filenames do not contain spaces. Fix it to handle filenames with spaces. □

Exercise 4-9. Compare the speed and space requirements of these versions of `bundle` and `unbundle` with variations that use headers and perhaps trailers. Evaluate the tradeoff between performance and program complexity. □

4.4 Multiline Records

The examples so far have featured data where each record fits neatly on one line. Many other kinds of data, however, come in multiline chunks. Examples include address lists:

```
Adam Smith
1234 Wall St., Apt. 5C
New York, NY 10021
212 555-4321
```

or bibliographic citations:

```
Donald E. Knuth
The Art of Computer Programming
Volume 4B: Combinatorial Algorithms, Part 2
Addison-Wesley, Reading, Mass.
2022
```

or personal databases:

```
Chateau Lafite Rothschild 1947
12 bottles at 12.95
```

It's easy to create and maintain such information if it's of modest size and regular structure; in effect, each record is the equivalent of an index card. Dealing with such data in Awk requires only a bit more work than single-line data does; we'll show several approaches.