1 AN AWK TUTORIAL

Awk is a convenient and expressive programming language that can be applied to a wide variety of computing and data-manipulation tasks. This chapter is a tutorial, designed to let you start writing your own programs as quickly as possible. Chapter 2 describes the whole language, and the remaining chapters show how awk can be used to solve problems from many different areas. Throughout the book, we have tried to pick examples that you should find useful, interesting, and instructive.

1.1 Getting Started

Useful awk programs are often short, just a line or two. Suppose you have a file called `emp.data` that contains the name, pay rate in dollars per hour, and number of hours worked for your employees, one employee record per line, like this:

```
Beth     4.00     0
Dan      3.75     0
Kathy    4.00     10
Mark     5.00     20
Mary     5.50     22
Susie    4.25     18
```

Now you want to print the name and pay (rate times hours) for everyone who worked more than zero hours. This is the kind of job that awk is meant for, so it's easy. Just type this command line:

```
awk '$3 > 0 { print $1, $2 * $3 }' emp.data
```

You should get this output:

```
Kathy 40
Mark 100
Mary 121
Susie 76.5
```

This command line tells the system to run awk, using the program inside the

quote characters, taking its data from the input file `emp.data`. The part inside the quotes is the complete awk program. It consists of a single *pattern-action statement*. The pattern, `$3 > 0`, matches every input line in which the third column, or *field*, is greater than zero, and the action

```
{ print $1, $2 * $3 }
```

prints the first field and the product of the second and third fields of each matched line.

If you want to print the names of those employees who did not work, type this command line:

```
awk '$3 == 0 { print $1 }' emp.data
```

Here the pattern, `$3 == 0`, matches each line in which the third field is equal to zero, and the action

```
{ print $1 }
```

prints its first field.

As you read this book, try running and modifying the programs that are presented. Since most of the programs are short, you'll quickly get an understanding of how awk works. On a Unix system, the two transactions above would look like this on the terminal:

```
$ awk '$3 > 0 { print $1, $2 * $3 }' emp.data
Kathy 40
Mark 100
Mary 121
Susie 76.5
$ awk '$3 == 0 { print $1 }' emp.data
Beth
Dan
$
```

The `$` at the beginning of a line is the prompt from the system; it may be different on your machine.

The Structure of an AWK Program

Let's step back a moment and look at what is going on. In the command lines above, the parts between the quote characters are programs written in the awk programming language. Each awk program in this chapter is a sequence of one or more pattern-action statements:

```
pattern   { action }
pattern   { action }
...
```

The basic operation of awk is to scan a sequence of input lines one after another, searching for lines that are *matched* by any of the patterns in the program. The precise meaning of the word "match" depends on the pattern in

question; for patterns like `$3 > 0`, it means "the condition is true."

Every input line is tested against each of the patterns in turn. For each pattern that matches, the corresponding action (which may involve multiple steps) is performed. Then the next line is read and the matching starts over. This continues until all the input has been read.

The programs above are typical examples of patterns and actions.

```
$3 == 0   { print $1 }
```

is a single pattern-action statement; for every line in which the third field is zero, the first field is printed.

Either the pattern or the action (but not both) in a pattern-action statement may be omitted. If a pattern has no action, for example,

```
$3 == 0
```

then each line that the pattern matches (that is, each line for which the condition is true) is printed. This program prints the two lines from the `emp.data` file where the third field is zero:

```
Beth     4.00     0
Dan      3.75     0
```

If there is an action with no pattern, for example,

```
{ print $1 }
```

then the action, in this case printing the first field, is performed for every input line.

Since patterns and actions are both optional, actions are enclosed in braces to distinguish them from patterns.

Running an AWK Program

There are several ways to run an awk program. You can type a command line of the form

`awk` *'program'* *input files*

to run the *program* on each of the specified input files. For example, you could type

```
awk '$3 == 0 { print $1 }' file1 file2
```

to print the first field of every line of `file1` and `file2` in which the third field is zero.

You can omit the input files from the command line and just type

`awk` *'program'*

In this case awk will apply the *program* to whatever you type next on your terminal until you type an end-of-file signal (control-d on Unix systems). Here is a sample of a session on Unix:

```
$ awk '$3 == 0 { print $1 }'
Beth    4.00    0
Beth
Dan     3.75    0
Dan
Kathy   3.75    10
Kathy   3.75    0
Kathy
...
```

The **heavy** characters are what the computer printed.

This behavior makes it easy to experiment with awk: type your program, then type data at it and see what happens. We again encourage you to try the examples and variations on them.

Notice that the program is enclosed in single quotes on the command line. This protects characters like `$` in the program from being interpreted by the shell and also allows the program to be longer than one line.

This arrangement is convenient when the program is short (a few lines). If the program is long, however, it is more convenient to put it into a separate file, say `progfile`, and type the command line

```
awk -f progfile   optional list of input files
```

The `-f` option instructs awk to fetch the program from the named file. Any filename can be used in place of `progfile`.

Errors

If you make an error in an awk program, awk will give you a diagnostic message. For example, if you mistype a brace, like this:

```
awk '$3 == 0 [ print $1 }' emp.data
```

you will get a message like this:

```
awk: syntax error at source line 1
 context is
        $3 == 0 >>>  [ <<<
        extra }
        missing ]
awk: bailing out at source line 1
```

"Syntax error" means that you have made a grammatical error that was detected at the place marked by `>>>` `<<<`. "Bailing out" means that no recovery was attempted. Sometimes you get a little more help about what the error was, such as a report of mismatched braces or parentheses.

Because of the syntax error, awk did not try to execute this program. Some errors, however, may not be detected until your program is running. For example, if you attempt to divide a number by zero, awk will stop its processing and report the input line number and the line number in the program at which the division was attempted.

1.2 Simple Output

The rest of this chapter contains a collection of short, typical awk programs based on manipulation of the `emp.data` file above. We'll explain briefly what's going on, but these examples are meant mainly to suggest useful operations that are easy to do with awk — printing fields, selecting input, and transforming data. We are not showing everything that awk can do by any means, nor are we going into many details about the specific things presented here. But by the end of this chapter, you will be able to accomplish quite a bit, and you'll find it much easier to read the later chapters.

We will usually show just the program, not the whole command line. In every case, the program can be run either by enclosing it in quotes as the first argument of the `awk` command, as shown above, or by putting it in a file and invoking awk on that file with the `-f` option.

There are only two types of data in awk: numbers and strings of characters. The `emp.data` file is typical of this kind of information — a mixture of words and numbers separated by blanks and/or tabs.

Awk reads its input one line at a time and splits each line into fields, where, by default, a field is a sequence of characters that doesn't contain any blanks or tabs. The first field in the current input line is called `$1`, the second `$2`, and so forth. The entire line is called `$0`. The number of fields can vary from line to line.

Often, all we need to do is print some or all of the fields of each line, perhaps performing some calculations. The programs in this section are all of that form.

Printing Every Line

If an action has no pattern, the action is performed for all input lines. The statement `print` by itself prints the current input line, so the program

```
{ print }
```

prints all of its input on the standard output. Since `$0` is the whole line,

```
{ print $0 }
```

does the same thing.

Printing Certain Fields

More than one item can be printed on the same output line with a single `print` statement. The program to print the first and third fields of each input line is

```
{ print $1, $3 }
```

With `emp.data` as input, it produces

```
Beth 0
Dan 0
Kathy 10
Mark 20
Mary 22
Susie 18
```

Expressions separated by a comma in a `print` statement are, by default, separated by a single blank when they are printed. Each line produced by `print` ends with a newline character. Both of these defaults can be changed; we'll show how in Chapter 2.

NF, the Number of Fields

It might appear you must always refer to fields as `$1`, `$2`, and so on, but any expression can be used after `$` to denote a field number; the expression is evaluated and its numeric value is used as the field number. Awk counts the number of fields in the current input line and stores the count in a built-in variable called `NF`. Thus, the program

```
{ print NF, $1, $NF }
```

prints the number of fields and the first and last fields of each input line.

Computing and Printing

You can also do computations on the field values and include the results in what is printed. The program

```
{ print $1, $2 * $3 }
```

is a typical example. It prints the name and total pay (rate times hours) for each employee:

```
Beth 0
Dan 0
Kathy 40
Mark 100
Mary 121
Susie 76.5
```

We'll show in a moment how to make this output look better.

Printing Line Numbers

Awk provides another built-variable, called `NR`, that counts the number of lines read so far. We can use `NR` and `$0` to prefix each line of `emp.data` with its line number:

```
{ print NR, $0 }
```

The output looks like this:

```
1 Beth  4.00    0
2 Dan   3.75    0
3 Kathy 4.00    10
4 Mark  5.00    20
5 Mary  5.50    22
6 Susie 4.25    18
```

Putting Text in the Output

You can also print words in the midst of fields and computed values:

```
{ print "total pay for", $1, "is", $2 * $3 }
```

prints

```
total pay for Beth is 0
total pay for Dan is 0
total pay for Kathy is 40
total pay for Mark is 100
total pay for Mary is 121
total pay for Susie is 76.5
```

In the `print` statement, the text inside the double quotes is printed along with the fields and computed values.

1.3 Fancier Output

The `print` statement is meant for quick and easy output. To format the output exactly the way you want it, you may have to use the `printf` statement. As we shall see in Section 2.4, `printf` can produce almost any kind of output, but in this section we'll only show a few of its capabilities.

Lining Up Fields

The `printf` statement has the form

`printf(`*format*, *value*$_1$, *value*$_2$, ... , *value*$_n$`)`

where *format* is a string that contains text to be printed verbatim, interspersed with specifications of how each of the values is to be printed. A specification is a % followed by a few characters that control the format of a *value*. The first specification tells how *value*$_1$ is to be printed, the second how *value*$_2$ is to be printed, and so on. Thus, there must be as many % specifications in *format* as *values* to be printed.

Here's a program that uses `printf` to print the total pay for every employee:

```
{ printf("total pay for %s is $%.2f\n", $1, $2 * $3) }
```

The specification string in the `printf` statement contains two % specifications.

The first, `%s`, says to print the first value, `$1`, as a string of characters; the second, `%.2f`, says to print the second value, `$2*$3`, as a number with 2 digits after the decimal point. Everything else in the specification string, including the dollar sign, is printed verbatim; the `\n` at the end of the string stands for a newline, which causes subsequent output to begin on the next line. With `emp.data` as input, this program yields:

```
total pay for Beth is $0.00
total pay for Dan is $0.00
total pay for Kathy is $40.00
total pay for Mark is $100.00
total pay for Mary is $121.00
total pay for Susie is $76.50
```

With `printf`, no blanks or newlines are produced automatically; you must create them yourself. Don't forget the `\n`.

Here's another program that prints each employee's name and pay:

```
{ printf("%-8s $%6.2f\n", $1, $2 * $3) }
```

The first specification, `%-8s`, prints a name as a string of characters left-justified in a field 8 characters wide. The second specification, `%6.2f`, prints the pay as a number with two digits after the decimal point, in a field 6 characters wide:

```
Beth     $  0.00
Dan      $  0.00
Kathy    $ 40.00
Mark     $100.00
Mary     $121.00
Susie    $ 76.50
```

We'll show lots more examples of `printf` as we go along; the full story is in Section 2.4.

Sorting the Output

Suppose you want to print all the data for each employee, along with his or her pay, sorted in order of increasing pay. The easiest way is to use awk to prefix the total pay to each employee record, and run that output through a sorting program. On Unix, the command line

```
awk '{ printf("%6.2f  %s\n", $2 * $3, $0) }' emp.data | sort
```

pipes the output of awk into the `sort` command, and produces:

```
  0.00   Beth    4.00    0
  0.00   Dan     3.75    0
 40.00   Kathy   4.00    10
 76.50   Susie   4.25    18
100.00   Mark    5.00    20
121.00   Mary    5.50    22
```

1.4 Selection

Awk patterns are good for selecting interesting lines from the input for further processing. Since a pattern without an action prints all lines matching the pattern, many awk programs consist of nothing more than a single pattern. This section gives some examples of useful patterns.

Selection by Comparison

This program uses a comparison pattern to select the records of employees who earn $5.00 or more per hour, that is, lines in which the second field is greater than or equal to 5:

```
$2 >= 5
```

It selects these lines from `emp.data`:

```
Mark    5.00    20
Mary    5.50    22
```

Selection by Computation

The program

```
$2 * $3 > 50 { printf("$%.2f for %s\n", $2 * $3, $1) }
```

prints the pay of those employees whose total pay exceeds $50:

```
$100.00 for Mark
$121.00 for Mary
$76.50 for Susie
```

Selection by Text Content

Besides numeric tests, you can select input lines that contain specific words or phrases. This program prints all lines in which the first field is `Susie`:

```
$1 == "Susie"
```

The operator `==` tests for equality. You can also look for text containing any of a set of letters, words, and phrases by using patterns called *regular expressions*. This program prints all lines that contain `Susie` anywhere:

```
/Susie/
```

The output is this line:

```
Susie    4.25    18
```

Regular expressions can be used to specify much more elaborate patterns; Section 2.1 contains a full discussion.

Combinations of Patterns

Patterns can be combined with parentheses and the logical operators &&, ||, and !, which stand for AND, OR, and NOT. The program

```
$2 >= 4 || $3 >= 20
```

prints those lines where $2 is at least 4 *or* $3 is at least 20:

```
Beth     4.00     0
Kathy    4.00     10
Mark     5.00     20
Mary     5.50     22
Susie    4.25     18
```

Lines that satisfy both conditions are printed only once. Contrast this with the following program, which consists of two patterns:

```
$2 >= 4
$3 >= 20
```

This program prints an input line twice if it satisfies both conditions:

```
Beth     4.00     0
Kathy    4.00     10
Mark     5.00     20
Mark     5.00     20
Mary     5.50     22
Mary     5.50     22
Susie    4.25     18
```

Note that the program

```
!($2 < 4 && $3 < 20)
```

prints lines where it is *not* true that $2 is less than 4 *and* $3 is less than 20; this condition is equivalent to the first one above, though perhaps less readable.

Data Validation

There are always errors in real data. Awk is an excellent tool for checking that data has reasonable values and is in the right format, a task that is often called *data validation*.

Data validation is essentially negative: instead of printing lines with desirable properties, one prints lines that are suspicious. The following program uses

comparison patterns to apply five plausibility tests to each line of `emp.data`:

```
NF != 3    { print $0, "number of fields is not equal to 3" }
$2 < 3.35  { print $0, "rate is below minimum wage" }
$2 > 10    { print $0, "rate exceeds $10 per hour" }
$3 < 0     { print $0, "negative hours worked" }
$3 > 60    { print $0, "too many hours worked" }
```

If there are no errors, there's no output.

BEGIN and END

The special pattern `BEGIN` matches before the first line of the first input file is read, and `END` matches after the last line of the last file has been processed. This program uses `BEGIN` to print a heading:

```
BEGIN { print "NAME     RATE    HOURS";  print "" }
      { print }
```

The output is:

```
NAME     RATE    HOURS

Beth     4.00     0
Dan      3.75     0
Kathy    4.00     10
Mark     5.00     20
Mary     5.50     22
Susie    4.25     18
```

You can put several statements on a single line if you separate them by semicolons. Notice that `print ""` prints a blank line, quite different from just plain `print`, which prints the current input line.

1.5 Computing with AWK

An action is a sequence of statements separated by newlines or semicolons. You have already seen examples in which the action was a single `print` statement. This section provides examples of statements for performing simple numeric and string computations. In these statements you can use not only the built-in variables like `NF`, but you can create your own variables for performing calculations, storing data, and the like. In awk, user-created variables are not declared.

Counting

This program uses a variable `emp` to count employees who have worked more than 15 hours:

```
$3 > 15 { emp = emp + 1 }
END     { print emp, "employees worked more than 15 hours" }
```

For every line in which the third field exceeds 15, the previous value of emp is incremented by 1. With emp.data as input, this program yields:

```
3 employees worked more than 15 hours
```

Awk variables used as numbers begin life with the value 0, so we didn't need to initialize emp.

Computing Sums and Averages

To count the number of employees, we can use the built-in variable NR, which holds the number of lines read so far; its value at the end of all input is the total number of lines read.

```
END { print NR, "employees" }
```

The output is:

```
6 employees
```

Here is a program that uses NR to compute the average pay:

```
    { pay = pay + $2 * $3 }
END { print NR, "employees"
      print "total pay is", pay
      print "average pay is", pay/NR
    }
```

The first action accumulates the total pay for all employees. The END action prints

```
6 employees
total pay is 337.5
average pay is 56.25
```

Clearly, printf could be used to produce neater output. There's also a potential error: in the unlikely case that NR is zero, the program will attempt to divide by zero and thus will generate an error message.

Handling Text

One of the strengths of awk is its ability to handle strings of characters as conveniently as most languages handle numbers. Awk variables can hold strings of characters as well as numbers. This program finds the employee who is paid the most per hour:

```
$2 > maxrate { maxrate = $2; maxemp = $1 }
END { print "highest hourly rate:", maxrate, "for", maxemp }
```

It prints

```
highest hourly rate: 5.50 for Mary
```

In this program the variable `maxrate` holds a numeric value, while the variable `maxemp` holds a string. (If there are several employees who all make the same maximum pay, this program finds only the first.)

String Concatenation

New strings may be created by combining old ones; this operation is called *concatenation*. The program

```
     { names = names $1 " " }
END { print names }
```

collects all the employee names into a single string, by appending each name and a blank to the previous value in the variable `names`. The value of `names` is printed by the `END` action:

```
Beth Dan Kathy Mark Mary Susie
```

The concatenation operation is represented in an awk program by writing string values one after the other. At every input line, the first statement in the program concatenates three strings: the previous value of `names`, the first field, and a blank; it then assigns the resulting string to `names`. Thus, after all input lines have been read, `names` contains a single string consisting of the names of all the employees, each followed by a blank. Variables used to store strings begin life holding the null string (that is, the string containing no characters), so in this program `names` did not need to be explicitly initialized.

Printing the Last Input Line

Although `NR` retains its value in an `END` action, `$0` does not. The program

```
     { last = $0 }
END { print last }
```

is one way to print the last input line:

```
Susie    4.25    18
```

Built-in Functions

We have already seen that awk provides built-in variables that maintain frequently used quantities like the number of fields and the input line number. Similarly, there are built-in functions for computing other useful values. Besides arithmetic functions for square roots, logarithms, random numbers, and the like, there are also functions that manipulate text. One of these is `length`, which counts the number of characters in a string. For example, this program computes the length of each person's name:

```
{ print $1, length($1) }
```

The result:

```
Beth 4
Dan 3
Kathy 5
Mark 4
Mary 4
Susie 5
```

Counting Lines, Words, and Characters

This program uses `length`, `NF`, and `NR` to count the number of lines, words, and characters in the input. For convenience, we'll treat each field as a word.

```
        { nc = nc + length($0) + 1
          nw = nw + NF
        }
END     { print NR, "lines,", nw, "words,", nc, "characters" }
```

The file `emp.data` has

```
6 lines, 18 words, 77 characters
```

We have added one for the newline character at the end of each input line, since `$0` doesn't include it.

1.6 Control-Flow Statements

Awk provides an `if-else` statement for making decisions and several statements for writing loops, all modeled on those found in the C programming language. They can only be used in actions.

If-Else Statement

The following program computes the total and average pay of employees making more than $6.00 an hour. It uses an `if` to defend against division by zero in computing the average pay.

```
$2 > 6 { n = n + 1; pay = pay + $2 * $3 }
END    { if (n > 0)
             print n, "employees, total pay is", pay,
                      "average pay is", pay/n
         else
             print "no employees are paid more than $6/hour"
       }
```

The output for `emp.data` is:

```
no employees are paid more than $6/hour
```

In the `if-else` statement, the condition following the `if` is evaluated. If it is true, the first `print` statement is performed. Otherwise, the second `print` statement is performed. Note that we can continue a long statement over several lines by breaking it after a comma.

While Statement

A `while` statement has a condition and a body. The statements in the body are performed repeatedly while the condition is true. This program shows how the value of an amount of money invested at a particular interest rate grows over a number of years, using the formula $value = amount\ (1 + rate)^{years}$.

```
# interest1 - compute compound interest
#   input:  amount  rate  years
#   output: compounded value at the end of each year

{   i = 1
    while (i <= $3) {
        printf("\t%.2f\n", $1 * (1 + $2) ^ i)
        i = i + 1
    }
}
```

The condition is the parenthesized expression after the `while`; the loop body is the two statements enclosed in braces after the condition. The `\t` in the `printf` specification string stands for a tab character; the `^` is the exponentiation operator. Text from a `#` to the end of the line is a *comment*, which is ignored by awk but should be helpful to readers of the program who want to understand what is going on.

You can type triplets of numbers at this program to see what various amounts, rates, and years produce. For example, this transaction shows how $1000 grows at 6% and 12% compound interest for five years:

```
$ awk -f interest1
1000 .06 5
        1060.00
        1123.60
        1191.02
        1262.48
        1338.23
1000 .12 5
        1120.00
        1254.40
        1404.93
        1573.52
        1762.34
```

For Statement

Another statement, `for`, compresses into a single line the initialization, test, and increment that are part of most loops. Here is the previous interest computation with a `for`:

```
# interest2 - compute compound interest
#   input:  amount  rate  years
#   output: compounded value at the end of each year

{   for (i = 1; i <= $3; i = i + 1)
        printf("\t%.2f\n", $1 * (1 + $2) ^ i)
}
```

The initialization `i = 1` is performed once. Next, the condition `i <= $3` is tested; if it is true, the `printf` statement, which is the body of the loop, is performed. Then the increment `i = i + 1` is performed after the body, and the next iteration of the loop begins with another test of the condition. The code is more compact, and since the body of the loop is only a single statement, no braces are needed to enclose it.

1.7 Arrays

Awk provides arrays for storing groups of related values. Although arrays give awk considerable power, we will show only a simple example here. The following program prints its input in reverse order by line. The first action puts the input lines into successive elements of the array `line`; that is, the first line goes into `line[1]`, the second line into `line[2]`, and so on. The `END` action uses a `while` statement to print the lines from the array from last to first:

```
# reverse - print input in reverse order by line

    { line[NR] = $0 }   # remember each input line

END { i = NR                # print lines in reverse order
      while (i > 0) {
          print line[i]
          i = i - 1
      }
    }
```

With `emp.data`, the output is

```
Susie   4.25   18
Mary    5.50   22
Mark    5.00   20
Kathy   4.00   10
Dan     3.75   0
Beth    4.00   0
```

Here is the same example with a `for` statement:

```
# reverse - print input in reverse order by line

    { line[NR] = $0 }   # remember each input line

END { for (i = NR; i > 0; i = i - 1)
          print line[i]
    }
```

1.8 A Handful of Useful "One-liners"

Although awk can be used to write programs of some complexity, many useful programs are not much more complicated than what we've seen so far. Here is a collection of short programs that you might find handy and/or instructive. Most are variations on material already covered.

1. Print the total number of input lines:

```
END { print NR }
```

2. Print the tenth input line:

```
NR == 10
```

3. Print the last field of every input line:

```
{ print $NF }
```

4. Print the last field of the last input line:

```
    { field = $NF}
END { print field }
```

5. Print every input line with more than four fields:

```
NF > 4
```

6. Print every input line in which the last field is more than 4:

```
$NF > 4
```

7. Print the total number of fields in all input lines:

```
    { nf = nf + NF }
END { print nf }
```

8. Print the total number of lines that contain `Beth`:

```
/Beth/ { nlines = nlines + 1 }
END    { print nlines }
```

9. Print the largest first field and the line that contains it (assumes some $1 is positive):

```
$1 > max { max = $1; maxline = $0 }
END      { print max, maxline }
```

10. Print every line that has at least one field:

```
NF > 0
```

11. Print every line longer than 80 characters:

```
length($0) > 80
```

12. Print the number of fields in every line followed by the line itself:

```
{ print NF, $0 }
```

13. Print the first two fields, in opposite order, of every line:

```
{ print $2, $1 }
```

14. Exchange the first two fields of every line and then print the line:

```
{ temp = $1; $1 = $2; $2 = temp; print }
```

15. Print every line with the first field replaced by the line number:

```
{ $1 = NR; print }
```

16. Print every line after erasing the second field:

```
{ $2 = ""; print }
```

17. Print in reverse order the fields of every line:

```
{ for (i = NF; i > 0; i = i - 1) printf("%s ", $i)
  printf("\n")
}
```

18. Print the sums of the fields of every line:

```
{ sum = 0
  for (i = 1; i <= NF; i = i + 1) sum = sum + $i
  print sum
}
```

19. Add up all fields in all lines and print the sum:

```
  { for (i = 1; i <= NF; i = i + 1) sum = sum + $i }
END { print sum }
```

20. Print every line after replacing each field by its absolute value:

```
{ for (i = 1; i <= NF; i = i + 1) if ($i < 0) $i = -$i
  print
}
```

1.9 What Next?

You have now seen the essentials of awk. Each program in this chapter has been a sequence of pattern-action statements. Awk tests every input line against the patterns, and when a pattern matches, performs the corresponding action. Patterns can involve numeric and string comparisons, and actions can include computation and formatted printing. Besides reading through your input files automatically, awk splits each input line into fields. It also provides a number of built-in variables and functions, and lets you define your own as well. With this combination of features, quite a few useful computations can be expressed by short programs — many of the details that would be needed in another language are handled implicitly in an awk program.

The rest of the book elaborates on these basic ideas. Since some of the examples are quite a bit bigger than anything in this chapter, we encourage you strongly to begin writing programs as soon as possible. This will give you familiarity with the language and make it easier to understand the larger programs. Furthermore, nothing answers questions so well as some simple experiments. You should also browse through the whole book; each example conveys something about the language, either about how to use a particular feature, or how to create an interesting program.