Basics of the *awk* Programming Language

Introduction

We now turn to another Unix tool, the *awk* programming language. I have chosen this language as the first one we will discuss because it has relatively few features – but still enough to be useful: I certainly use it for lots of the "small tasks" that often turn up. Also, you can learn *awk* more quickly because it is an "interpreted" language; the *awk* processor reads what you write and executes it as it reads. The cycle of writing a program and fixing errors thus is relatively quick; since learning a programming language, and developing code, is largely about finding errors, you learn the language faster.

A brief digression on language types: many computer languages are not interpreted. Rather, what happens is that first they are converted from what you write to what the computer executes by by a *compiler*. In the simplest case, the compiler produces *object code* that can be executed by the computer directly; more often, this code is first *linked* with object code for other programs to produce the final *executable*. The object code, being directly interpretable by the computer, is very fast; but the step of first compiling the object code itself takes time. Interpreted languages run much more slowly, but this doesn't matter for small tasks, where the computation time will be much less than the time you spend on development.

These notes, like the others I have prepared, are not complete; *awk* has many features I will not discuss. The goal is to provide enough that you can do useful things with *awk* programs – and more importantly, enough for you to get the feel of what it is like to write programs. Section describes some of the things that I do not cover and that you might want to learn about,

A First Example

Most introductions to *awk* start with its ability to match and print patterns, which makes it a more powerful version of *sed*. I will instead start with a numerical example, as it is easier to see what is being done. For historical reasons[1] angles are often written in sexagesimal notation, in degrees, minutes, and seconds, with the last being decimalized: for example, $23°41'18''$. But most trigonometric functions take arguments in either decimal degrees or radians. The conversion is just arithmetic, which we can program in *awk* as:

[1] Namely, that the Babylonians used base-60 notation, and also were the first astronomers, who used angles to describe where things were.

```
BEGIN{dr=3.14159265/180.}
{
  d=$1+($2+$3/60)/60
  print d,dr*d
}
```

(we use `this font` for program code). If we put this in a file called `baby`[2] we could then run it as follows:

```
% awk -f baby
23 41 18
23.6883 0.413439
12 34 56
12.5822 0.219601
60 60 60
61.0167 1.06494
1 0 0
1 0.0174533
0 0 0
0 0
1 30 0
1.5 0.0261799
-1 0 0
-1 -0.0174533
-1 30 0
-0.5 -0.00872665
%
```

What is going on here? The shell invokes the *awk* interpreter, for which a `-f` flag means "what follows is the name of a file of *awk* commands to be interpreted". The first thing the interpreter sees is a `BEGIN{`, which means "run the following commands until you see a `}`". There is only the one command, which sets the variable `dr`, used for conversion from degrees to radians.[3] The second pair of braces mean "read in a line from standard input, perform the commands between the braces, and repeat this until there are no more lines". Typing at the terminal, we signal "no more lines" by hitting the Cntrl-D key.

For each line, *awk* assigns each field (anything surrounded by white space, the start of the line, or the end of the line) to a variable; the n-th field is

[2] The name is derived from the history just mentioned.

[3] The `=` sign means "assign what is on the right to the variable on the left".

referenced in the program by the variable `$n`. These variables can be character strings or numbers; *awk* decides what they are depending on what you do with them. So, when our first line is typed in, `$1` is 23, `$2` is 41, and so on. The next commands do the arithmetic, assigning values to `d` and `dr*d`, and then printing these values out as `23.6883 0.413439`. Subsequent lines are handled the same way. Note that we write the expression as `$1 + ($2+$3/60)/60`, using what is called a nested arrangement, with parentheses to set the proper order of operations.

Control Flow

Looking at the various inputs and outputs, we see one oddity and one error. The oddity is that the program is perfectly happy to accept expressions such as `60 60 60`, which in proper notation would be $61°01'00''$. But since we get the right answer we can ignore this as harmless. However, the program does not handle negative values properly. An angle of $-1°30'00''$ should be interpreted with the minutes and seconds having the same sign as the degrees; the computer, as it does all too often, has done what we asked for, not what we wanted. This is not difficult to fix, and introduces an example of *control flow*:

```
BEGIN{dr=3.14159265/180.}
{
  if($1>=0) d=$1+($2+$3/60)/60
  if($1<0)  d=$1-($2+$3/60)/60
  print d,dr*d
}
```

which gives:

```
% awk -f baby
-1 0 0
-1 -0.0174533
-1 0 0
-1.5 -0.0261799
%
```

The statement

`if(`*something*`)`

causes what follows it to be executed if *something* is assigned the value of `TRUE`. "What follows" can be a single statement, as shown above, or a number of statements held together by braces; the *awk* program could as well be written as

```
BEGIN{dr=3.14159265/180.}
{
  if($1>=0) {
     d=$1+($2+$3/60)/60
  }
  if($1<0) {
     d=$1-($2+$3/60)/60
  }
  print d,dr*d
}
```

The indentation in front of each line is meaningless to the computer – but it is well worth your while to use it to make the program more obvious to you, for example by indenting groups of statements, as shown above, rather than by writing (for example)

```
BEGIN{dr=3.14159265/180.}{if($1>=0) d=$1+($2+$3/60)/60
if($1<0) {
d=$1-($2+$3/60)/60
}
print d,dr*d}
```

which is the same to the computer, but harder to read.

Truth and Falsity

It is worthwhile to go a little deeper into the part between the parentheses in the `if()` statement. Again, what happens here is that the *awk* interpreter looks at what is there and evaluates it; we can think of the material between the parentheses as an expression, just like an expression in arithmetic, but whose value is either `TRUE` or `FALSE`: such an expression is called a **logical variable**. A statement such as `if(d>0){` is equivalent to

```
yn = (d>0)
if(yn){
```

We need to first consider what form the expression must have to be acceptable to *awk*. arithmetic expressions are acceptable (that is, can be understood by *awk*) if they make sense algebraically:

```
((3.14*a+b**4)/c) + d - x/v
```

makes sense[4] but `x*+y` does not.

So, what are acceptable forms for truth-valued statements? The general form is variables connected by **logical operators**. The available logical operators are listed in Table , and described in terms of what makes the overall expression true, given the logical values (truth or falsity), or relative numerical or other values of the variables on either side of the expression. The first four operators in Table are basically the same as the algebraic ones. Note that these can, in principle, be applied to other kinds of variables than numerical values, though unless you know what you are doing this is not a good idea; for example, is the character string `finagle` greater than `flange`?

The next two operators evaluate if the variables are the same or not. This might mean "have the same numerical value", but extends beyond this to, for example, pairs of character strings. Remember that "the same as"

[4] Remembering that `x**n` is what is written as x^n in algebraic notation.

Operator	Name	Meaning
`>`	greater than	True if previous variable exceeds following one
`<`	less than	True if following variable exceeds previous one
`>=`	greater than or equal to	True if previous variable exceeds or equals following one
`<=`	less than or equal to	True if following variable exceeds or equals previous one
`==`	equal to	True if previous variable equals following one
`!=`	not equal to	True if previous variable does not equal following one
`&&`	and	True if both flanking variables are true
`||`	or	True if either flanking variable is true

Table 1: Logical operators in *awk*

includes non-printing characters; " `and` " (with blanks) is not the same as "`and`" (without).

Notice that `==`, meaning "has the same value as" is not at all the same as `=`, which means "assign the value on the right to the variable on the left. In my experience, confusion between these is easy when typing, and can lead to quite mysterious behavior. Suppose you wanted `if(x==1)`, which means "do the following if `x` is `1`", but type `if(x=1)` by mistake. This will cause very different behavior, because `if(x=1)` evaluates the statement `x=1` in two ways: it assigns `x` the value `1`, and assigns the statement the value `TRUE` – so whatever the `if` controls, always happens, with `x` given a new, fixed value into the bargain. The stubborn refusal of the program to create different values can be very puzzling.

The last two operators are for logical variables; while we do not usually start with the such variables, they often occur as intermediate values in a compound statement. For example, `x>1` and `y<=0` both evaluate as logical variables, from which we could form a compound statement such as `(x>1)||(y<=0)`, which would be true if either substatement (the first or the last) was true, or `(x>1)&&(y<=0)`, which would be true only if both (the first and the last) were true.

Including *awk* in Scripts

So far we have assumed that the instructions for *awk* will be placed in a separate file, and that we will tell *awk* to read these instructions using `awk -f` *file*. But in fact we can include the instructions with the invocation of *awk*, either at the terminal or in a script; unless you are doing something simple, or never make typing errors, you should use a script. As an example, if we rewrite our `baby` script to include the invocation of *awk*, the file `baby` would be:

```
awk 'BEGIN{dr=3.14159265/180.}
{
  if($1>=0) d=$1+($2+$3/60)/60
  if($1<0) {
     d=$1-($2+$3/60)/60
  }
  print d,dr*d
}'
```

and we would run it by first making it executable with `chmod` and then running it:

```
% chmod +x baby
% baby
-1 0 0
-1 -0.0174533
-1 0 0
-1.5 -0.0261799
%
```

Including the invocation and the commands in the same file is best: you don't lose anything by including the invocation of *awk* in the file. Since you will need to do so if there is more than one invocation of *awk* in the script, it is just as well to make a habit of writing *awk* into any script that uses it. In these notes I will generally not do this, but that is only to save space.

One question that is best addressed here is how to put a shell variable (from the command line of the script) into *awk*: to do so, equate a variable name in *awk* to a shell variable just after the *awk* invocation. For example, we can modify the above script to work in some other base than 60 by writing it as

```
awk 'BEGIN{dr=3.14159265/180.}
{
  if($1>=0) d=$1+($2+$3/base)/base
  if($1<0) {
     d=$1-($2+$3/base)/base
  }
  print d,dr*d
}' base=$1
```

which will put whatever you type as the first command-line argument into the variable `base`. With this modification, we could accomplish the totally pointless task of going from decimal to decimal:

```
% baby 10
123 4 5
123.45 2.15461
%
```

Note that the variable cannot be used in the part of the *awk* program just after the `BEGIN`; it is only visible in the main loop and after. (There is a way around this, but you'll have to look it up).

Variables

Up to now I may seem to have been vague about what kinds of variables there are. This is because, though *awk* has two types of variables, it is very permissive about assigning them. You can pretty much use any name for any type of variable. The main types of interest are:

A. **Numeric:** these are just like variables in algebra; they are assumed to be real numbers, though with finite precision.

B. **String:** these are strings of characters. The string `9` is distinct from the numeric value 9; writing `x = 9` assigns a numeric value to `x`, while writing `x ="9"` assigns the single character `9` to `x`.

C. **Logical:** we talked about these earlier, but I should now reveal, for the sake of accuracy, that *awk* does not actually have such a variable type; instead `TRUE` is represented by any non-zero numeric value, or any non-empty string value; `FALSE` is represented by zero (numeric) or the empty string `""`.

Obviously, there is plenty of opportunity to get into trouble if you try to mix numeric and string variables – which you can do. And *awk* will decide what type a variable is depending on what you do with it. For example, consider the following program fragment:

```
x="SIO"
y=233
z=x+y
print x,y,z
y="233"
z=x+y
print x,y,z
z=x+1+y+1
print x+1,y+1,z
```

which produces the output

```
SIO 233 233
SIO 233 233
1 234 235
```

showing that the string "`233`" is converted to the numeric value `233` when it is included in an addition.

So far I have discussed variables that you, the programmer, are free to make up and assign. But *awk* also has preset variables that relate to the lines it reads in. Most of these are strings, but again could be interpreted as numeric if appropriate. As noted in Section , the variable `$n` means "the n-th field on the input line"; the variable `$0` means "the entire input line". A **field** is a string surrounded by **field separators**; by default these are blanks and tabs (plus the start and end of the line). The variable `NF` is the number of fields on a line; the variable `NR` (the "record number") is the number of the line read in, starting with one. So an *awk* program to print out lines 25 through 44 would be `{if(NR>=25&&NR<=44) print $0}`. The final named variable you are likely to need is `FS`, which contains the value of the field separator. If, for example, you wanted to treat as separate fields strings that contained blanks, but were separated by tabs, you would write `FS="TAB"` at the start of the program (in the part done by the `BEGIN` statement).[5]

Numeric Operations and Functions

Though *awk* was originally intended more for dealing with character strings than with numbers, it has acquired enough mathematical capability to be useful for simple calculations. In addition to the usual arithmetic operators, `+`, `-`, `*`, and `/`, we have already encountered exponentiation `**`. Another arithmetic operator is `%`, which means "produce the remainder after dividing the previous variable by the following one". So, for example, `1%2` is 1, `2%1` is 0, `99%9` is 0, `99%7` is 1, `2.1%1` is 0.1, `3.2%2` is 1.2, `10.3%3` is 1.3, `10.3%5` is 0.3, `1234567895%5` is 0, `10.3%.1` is 1.38778×10^{-16}, and `12345678876543215%5` is 1. The last two cases show that the internal arithmetic is not infinitely precise: the answers should both be zero.

A function related to the above is `int(x)`, which returns the integer part of `x`, an operation usually called **truncation**. The truncation is always towards zero, so `int(1.2)` is 1, `int(0.5)` is 0, `int(0.999)` is 0, `int(100.3)` is 100, `int(-1.2)` is -1, `int(-3.999)` is -3, and `int(-0.1)` is 0.

Built-in functions in *awk* include the usual elementary ones: the square root `sqrt`, the exponential `exp`, the natural log `log`, the sine and cosine `sin` and `cos`, and the two argument arctangent `atan2`. The last one, if called as (say) `atan2(y,x)`, evaluates the arctangent of y/x, but keeping the signs to get the correct angle between 0 and 2π. So, for example, `atan2(0.0,1.0)` is $0°$, `atan2(0.5,1.0)` is $26.565°$, `atan2(1.0,1.0)` is $45°$, `atan2(1.0,0.5)` is

[5] Of course, you would press the TAB key, not type the letters TAB.

63.435°, `atan2(1.0,0.0)` is 90°, `atan2(-1.0,0.0)` is -90°, `atan2(0.0,-1.0)` is 180°, `atan2(-1.0,-1.0)` is -135°, `atan2(1.0,-1.0)` is 135°. (Actually, the values are returned in radians, but I have converted them to degrees for clarity.)

Finally, *awk* includes a random-number generator, or more correctly a **pseudorandom** number generator, or PRNG. This is a function that, every time it is called, returns a numerical value between zero and one, computed so that successive values are unrelated to each other, and a large number of values will be uniformly distributed over their possible range. The reason for the "pseudo" is that these numbers are in fact completely deterministic: they are computed using arithmetic in such a way as to produce the properties just given. This means that if we had an *awk* program called `random`

```
{
 for(i=1;i<=$1;i++) {
   x=rand()
   print x
 }
}
```

every time we ran the line `echo 10 | awk -f random` we would get the same 10 numbers.[6] The way around this is to know that the sequence output by the PRNG is controlled by an initial value, called a **seed**. Setting different values of the seed will produce completely different sequences. The seed is set within *awk* using a function `srand`. Calling `srand(x)` will set the seed value to `x`; calling `srand()` will set the seed to a value that depends on the current date and time, and thus will be different every time you run the program.

Looping

We now return to control flow, to discuss the other main form of it, which is looping: that is, repetition of the same set of commands, over a specified range of some variable. Suppose (to take a very simple example) that we wanted to print out the first `n` powers of a given number. We assume that we will type in the number, and the number of powers; then the *awk* script (call it `power`) would be

```
{
```

[6] I get 0.237788, 0.291066, 0.845814, 0.152208, 0.585537, 0.193475, 0.810623, 0.173531, 0.484983, and 0.151863.

```
  for(i=1;i<=$2;i++) {
    print $1**i
  }
}
```

```
% awk -f power
2 4
2
4
8
16
3 8
3
9
27
81
243
729
2187
6561
2.5 5
2.5
6.25
15.625
39.0625
97.6562
%
```

here the `for(i=1;i<=$2;i++) {` statement is the start of the loop, which is closed by the `}`. The `for` statement has three parts, which set the starting value of the looping variable, the range of values for which the loop will be repeated, and the increment to the looping variable each time the loop is executed. The expression `i++` is equivalent to `i=i+1`. So the loop is executed until `i` is incremented to a value for which the middle statement is no longer true; then *awk* exits from the loop, and reads the next line of input: this reading of lines is an **implicit loop**, also called the **outer loop** of the program.

It may be appropriate, when using *awk* for numerical purposes, not to use this outer loop at all – or rather, to use it once, just to get the program to execute. Here, for example, is a script to write 1000 random numbers:

```
echo x |\
```

```
awk 'BEGIN{srand()}
{
  for(i=1;i<=1000;i++) print rand()
}'
```

Input and Output

Our examples up to now have used the `print` statement for output. But *awk* can also print in a variety of specified formats, using the `printf` statement. If you know C this will be familiar; if not, it won't.

The basic syntax of this statement is

`printf"`*formatting information*`",`*variable*`,`*variable*`,...`

where what is between the `"`'s is material that will be printed out literally, and **format specifications** for how the variables are to be printed. Everything that is not a format specifier will be printed as given. this literalness extends to the newline character; if you want this string at the end of a line, you must specify it by including a `\n` at the end of the formatting information. In fact, you can include `\n` anywhere in the formatting information; one `printf` statement can print more than one line.

The format specifiers all start with a `%`. The simplest is `%s`, which means "print as a string". So `printf"%s\n%s\n%s\n",s1,s2,s3` would print the three string variables on three lines. (The number of format specifiers must match the number of variables). If the `s` is preceded by a number, the string will print out using at least that many spaces; if `s1` is less than 20 characters long, `printf"%20s\n",s1` would print `s1`, after enough spaces to make up a total of 20. (If `s1` is more than 20 characters long, this specification would print it in full, with no additional spaces). Preceding the number by `-` (for example `%-20s`) makes the string left-justified instead.

There are a large number of ways to format numbers. The specifier `%d` will output the integer part of the variable; again, a number will give the number of spaces, and a `-` will cause left justification. So if you are writing integers of less than (say) 9 figures, a `%9f` will print them in a nicely justified column. If you precede the number with a zero, the value will be written out with leading zeros; for example, `%03d` would write the value 22 as `022`.

The specifier `f` will output the variable as a decimal value, with the number of decimal places being set by a number following a period. For example, `%.4f` means "print a value with four decimal places". In this case, any number before the period sets the total number of characters to print, and a `-`before that sets

the justification: `%-10.6f` will use output 10 characters at least, left-justified, and with 6 decimal places.

This is all probably better shown than outlined, so here is a program that uses a range of formats, followed by its output.

```
echo 3.14159265358979 |\
awk '{pi=$1
      s="pi and pi**40 are "
      pi40=pi**40
      print s,pi,pi40
      printf"%s %f %f\n",s,pi,pi40
      printf"%s %.4f %.4f\n",s,pi,pi40
      printf"%40s %.4f %.0f\n",s,pi,pi40
      printf"%-40s %.4f %.4f\n",s,pi,pi40
      printf"%s %.6f %d\n",s,pi,pi40
      printf"%s %.18f %.4f\n",s,pi,pi40
      printf"pi is %.4f and pi**40 is %.4e\n",pi,pi40
}'
```

```
pi and pi**40 are  3.14159265358979 7.69121e+19
pi and pi**40 are  3.141593 76912142205153984512.000000
pi and pi**40 are  3.1416 76912142205153984512.0000
                      pi and pi**40 are  3.1416 7691214220515398451 2
pi and pi**40 are                        3.1416 76912142205153984512.0000
pi and pi**40 are  3.141593 7.69121e+19
pi and pi**40 are  3.14159265358979007 76912142205153984512.0000
pi is 3.1416 and pi**40 is 7.6912e+19
```

String Operations and Functions

Section described a set of operations (arithmetic, mostly) and functions for numeric variables. But *awk* has similar capabilities for string-values variables. The only operation involving string variables is concatenation, which is done with no symbol at all; if we write

```
a = "Colorless green ideas "
b = "sleep furiously."
c = a b
```

`c` would be have the (string) value `Colorless green ideas sleep furiously.`

This has one unexpected consequence in using `print` statements; if you write `print $1 $2` these two variables will appear without a space between them; you must write `print $1,$2` to get a space.

More interesting are the various functions that take string-valued variables as arguments. The simplest are `toupper(`*string*`)` and `tolower(`*string*`)`, which return the string value with all the letters set to uppercase or lowercase respectively.

In my experience one string function that sees a lot of use is `substr`, which has the syntax

`a = substr(`*string*`,`*nstart*`,`*ntake*`)`

where *string* is a string-valued variable, and *nstart* and *ntake* are numeric ones (which may of course just be numbers). This function returns (passes to `a` in the example) a total of *ntake* characters from *string*, starting at the one numbered *nstart*. So, extending our example above

```
a = "Colorless green ideas "
b = "sleep furiously."
c = a b
d = substr(c,1,11)
e = substr(c,15,15)
```

would make `d` equal to `Colorless g` and `e` equal to `n ideas sleep f`.

Using the `substr` function on the *awk* variable `$0` (which represents the entire line read in) is a great way to get parts of a series of lines that appear in different columns, if the blanks cannot be used as separators. For example, lines 4 through 10 below come from the southern California earthquake catalog

```
         1         2         3         4         5         6         7         8
12345678901234567890123456789012345678901234567890123456789012345678901234567890
Year Mo Dy  Hr Mn  Sec     Lat     Long     Q Mag      Dep  Ns     Err   CID
1999 10 16  09 46 44.13  34 35.64-116 16.26 A 7.1      0.02 55     0.16 9108652
1999 10 16  09 47 43.76  33 13.92-115 39.60 D 4.7      6.00 11     0.32 3327063
1999 10 16  09 50 49.15  34 13.26-116 21.96 C 3.0      6.00 10     0.26 3327068
1999 10 16  09 51 48.33  34 26.71-116 15.82 A 4.8      0.26 51     0.10 3320846
1999 10 16  09 52 15.80  34 36.54-116 17.16 C 4.2      6.00 13     0.38 3327069
1999 10 16  09 52 53.97  34 30.12-116 12.18 C 5.0      6.00 22     0.29 3320847
1999 10 16  09 54 54.93  34 37.02-116 17.34 C 4.3      2.71 20     0.26 3327070
```

I have used the first three lines to show the numbering along the line, and the labels.[7] Unfortunately, there is no space between the minutes of latitude and the degrees of longitude; so to convert these coordinates to decimal we would need

`latd=$7`

[7] That the number of characters goes to 80 is a vestige of a much older technology, namely the 80-column punch card, introduced by IBM in 1928.

```
latm=substr($0,29,5)
lond=substr($0,34,4)
lonm=$9
lat=latd+latm/60
lon=lond-lonm/60
```

But what if you want to get all the characters from (say) number 13 to the end of the string? Then you need the `length` function, which is `length(`*string*`)`, and returns the number of characters in the string. With this, getting all the characters from 13 on, regardless of the string length, would be done with `substr(`*string*`,13,length(`*string*`)-12)`. You need the `-12` because the length is counted inclusively: if it were 13 characters long, you want one character, not zero.

The next two functions we consider are `index`, and `match`. The first one has the syntax

`n = index(`*string1*`,`*string2*`)`

which returns the character number for which *string2* matches *string1*, or zero if it does not. For example

```
a = "Colorless green ideas "
b = "sleep furiously."
c = a b
k = index(c,a)
n = index(c,b)
m = index(c,"green ideax"
o = index(c," (Noam Chomsky)")
```

would set `k` to one, `n` to 23, and `m` and `o` to zero. The second has the syntax

`n = match(`*string1*`,`*regexp*`)`

which returns the character number for which the regular expression finds a match to the string.

Arrays

We finish with a discussion of arrays: that is, sets of variables, each of which is associated with an index. The most familiar example of this is probably that of a vector, which is a collection of N numbers $x_1, x_2, ...x_N$, each associated with an integer from 1 to N. As you might imagine, awk allows both numeric

and string variables to be indexed, with the syntax being *var*[*index*]. So, for example, we can refer to variables in an array in statements such as

```
a[1] = 3.14**6
a[2] = 2
b = a[1]/a[2]
```

where the size of the array is set by how many indexed variables we want to refer to, with awk keeping track of this as it goes.

As an example, here is an awk program that reads in lines, each of which has some number of numbers on it, and returns the exponentials of these numbers, with no more than three on a line

```
awk '{
for(i=1;i<=NF;i++) {
  out[i]=exp($i)
}
for(i=1;i<=NF;i++) {
  printf"%20.3f ",out[i]
  if(i%3==0) printf"\n"
}
if(i%NF!=0) printf"\n"
}'
```

The first *for* loop loads the array; the next one prints it out. The `printf` statements are arranged to produce a space, with a newline only every three values; the last `printf` adds a newline at the end if one was not just printed out. If we use this script to read from a file that contains

```
1 2 3 4 5 6 7 8 9 10
0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
```

we get

```
               2.718               7.389              20.086
              54.598             148.413             403.429
            1096.633            2980.958            8103.084
           22026.466
               1.105               1.221               1.350
               1.492               1.649               1.822
               2.014               2.226               2.460
```

There is a string function `split` than can be useful in filling an array; this has the syntax

`n =`
`split(`*string*`,`*array*`,`*separator*`)`

which returns the number of elements in the *string* separated by the *separator* string, and puts those elements into *array*. Suppose, for example, that we have dates given (as in the earthquake catalog above) in the form of year, month, and day, and want to convert these to year and day of year. An awk script for doing this would be

```
awk 'BEGIN {split("0 31 59 90 120 151 181 212 243 273 304 334",mth)}
{
   yr=$1
   day=mth[$2]+$3
   if(yr%4==0&&mth[$2]>=59) day=day+1
   print yr, day
}'
```

The part executed before reading sets up an array of the number of the cumulative number of days before each month. The main loop computes the day of the year using this and the day of the month, with a correction for leap years.[8]

You might wonder if the indices in an array have to be numeric – and they do not. An array that uses strings for indexes is called an **associative array**, and can be very useful indeed. Here is a simple example, for converting dates of the form 13 May 2009 into a year and day of year:

```
awk 'BEGIN {
   month["Jan"] = 0
   month["Feb"] = 31
   month["Mar"] = 59
   month["Apr"] = 90
   month["May"] = 120
   month["Jun"] = 151
   month["Jul"] = 181
   month["Aug"] = 212
   month["Sep"] = 243
   month["Oct"] = 273
   month["Nov"] = 304
```

[8] It is convenient that, according to the Gregorian calendar, all years from 1801 through 2099 and divisible by 4 are leap years.

```
   month["Dec"] = 334
}
{
   yr=$3
   day=month[$2]+$1
   if(yr%4==0&&month[$2]>=59) day=day+1
   print yr, day
}
```

which is very much the same as the previous example, but shows how we may index by strings.

You may now be wondering how, if the index is not a number, it is possible to step through it as we can with numeric indices – it isn't, but awk provides a syntax that amounts to doing this. First of all, there is a logical expression

item `in` *array*

which is TRUE if *item* is one of the **indices** of *array*, and FALSE otherwise. For example, in the last example, `"Jul" in month` would be TRUE: (so `if("Jul" in month)` would cause something to happen), whereas `181 in month` would be FALSE.

Secondly, we can say

`for(` *variable* `in` *array* `)`

which will cause the *variable* to cycle through all the indices of the array. We could, for example, get the weather for each month[9]

```
awk 'BEGIN {
   month["Jan"] = "Ice and snow"
   month["Feb"] = "Hail and sleet"
   month["Mar"] = "Wintry wind"
   month["Apr"] = "Endless showers"
   month["May"] = "Frost and hail"
   month["Jun"] = "Rains and never stops"
   month["Jul"] = "Occasional bright intervals"
   month["Aug"] = "Cold and dank and wet"
   month["Sep"] = "Mist and mud"
   month["Oct"] = "Wind and slush and rain and hail"
```

[9] Modified from a song about the English weather, by Michael Flanders and Donald Swann, and the BBC weather forecasts.

```
   month["Nov"] = "Fog and dark"
   month["Dec"] = "Freezing wet"
}
{
   for($2 in month) {
     print month[$2]
   }
}'
```

Things Undiscussed

I have left out a great many things in this discussion. Here are a few that might be worth looking into:

1. **Control Flow.** There are a number of other kinds of loops, such as the `do-while`, that can be useful.

2. **Input and Output.** The `getline` statement provides a useful way to read information in from a file other than the one you are piping to the *awk* program.