An Introduction to Regular Expressions and Awk

Eric Kostelich

October 2, 2018

1 Regular expressions

A *regular expression* is a rule for matching text. There are many variations of regular expression syntax, and various efforts have been undertaken to standardize it. The discussion here describes a subset of the IEEE Standard 1003.1 (2004) for regular expressions.

Regular expressions consist of a mixture of *ordinary characters*, which match themselves, and *special characters*, which have a special meaning. Ordinary characters include all letters of the English alphabet, the decimal digits, and the ordinary space character. For instance, the letters `c`, `a`, and `t` all are ordinary characters, and the regular expression `cat` matches strings containing any of the words *cat*, *cat*s, tom*cat* and s*cat*ter.

Pattern matching with regular expressions uses the following simple rule: *If there is more than one potential match within a given string, the leftmost match is taken.* In the string "The catalyst produces cations", `cat` matches *cat*alyst. This rule is important when using the search-and-replace feature of text editors. There is no way (that I know of) to match, say, only the second occurrence of a regular expression within a string. To match the *cat* in *cation*, for example, a pattern like `cati` would be needed.

Special characters are used to denote matches with any of a range of possibilities. The idea is best illustrated by example:

`^cat`	matches strings that begin with *cat*
`cat$`	matches strings that end with *cat*
`^cat$`	matches lines containing only the 3-letter string *cat*
`c[aou]t`	matches *cat*, *cot*, and *cut*
`c.t`	matches any 3-character sequence starting with *c* and ending with t
`\.`	matches a literal period
`[A–Z]`	matches any upper-case letter
`^[a–z]`	matches any string that begins with a lower-case letter
`[0–9]$`	matches any string that ends with a decimal digit
`[0–9][0–9]`	matches any string containing two consecutive decimal digits
`[A–Za–z0–9]`	matches any alphanumeric character
`[^0–9]`	matches any character that is *not* a digit
`^[^0–9]`	matches any string that does not begin with a digit

These examples illustrate the so-called *basic* regular expressions; Table 1 contains a summary.

1.1 The grep utility

grep stands for General Regular Expression Printer; it is a standard Unix text processing utility that locates and prints lines that match basic regular expressions. Among other uses, grep is handy for locating variable names in program source code. The general command-line format is

grep *options pattern files*

If no file name appears, then grep reads from its standard input. By default, grep's output consists of precisely those lines in the input that match the basic regular expression *pattern*.

Example 1. Here are some example uses of grep:

grep xyz file.c	prints all lines in file.c containing the string *xyz*
grep –n xyz file.c	as above, but prepends the number of each matching line
grep –i xyz file.c	ignore case: match *XYZ*, *Xyz*, etc.
grep –l xyz *.c	print only the names of the .c files containing *xyz*
grep –v xyz file.c	prints all lines that do *not* match *xyz*

This is only a partial list of grep's options. Consult the online manual page (man grep) for a complete list and further details. □

Regular expressions containing special characters should be enclosed in single quotation marks: Special characters often have special meaning to the shell as well, and single quotes prevent the shell from interpreting them.

Example 2. Many regular expression special characters, such as dollar signs and square brackets, are significant to the shell. Place such expressions in single quotes to keep the shell from altering them:

grep 'xyz$' file.c	prints all lines in file.c that end with *xyz*
grep 'a[xyz]e' file.c	matches any lines containing *axe*, *aye*, or *aze* in file.c
grep 'xyz' file.c	same as grep xyz file.c

Although the last expression contains no special characters, it may be enclosed in single quotes just like the others. Write defensively: when in doubt, enclose regular expressions in single quotes. □

.	(period) matches any single character
^p	matches any string beginning with the pattern p
p\$	matches any string ending with the pattern p
^p\$	matches the string that is precisely p
[$c_1 \cdots c_n$]	matches any single character in the set $\{c_1, \ldots, c_n\}$.
[c_1–c_2]	matches any single character between c_1 and c_2, inclusive, in the lexicographic ordering
\\s	matches s, which otherwise would be special
[[:alnum:]]	any alphanumeric character
[[:alpha:]]	any alphabetic character
[[:cntrl:]]	any control character
[[:graph:]]	any printable character
[[:print:]]	any printable character or blank
[[:punct:]]	any punctuation character
[[:space:]]	any whitespace character (e.g., blank or tab)
[[:digit:]]	any decimal digit
[[:xdigit:]]	any hexadecimal digit
[[:upper:]]	any uppercase letter
[[:lower:]]	any lowercase letter

Table 1: The syntax of some basic regular expressions.

1.2 Locales and named character classes

One caveat applies to the use of ranges: the characters that are matched depend on the underlying character set that is being used for the computation. In an environment that uses the ASCII character set, the uppercase letters of the English alphabet form a contiguous increasing sequence, so the range [A–Z] suffices to denote a single uppercase letter. Such is not the case, however, with other character sets: the uppercase letters do *not* form a contiguous sequence in the EBCDIC character set that is commonly used on IBM mainframes, for example.

To reduce the dependency on the underlying character set, many implementations of regular expression parsers (and the programs that use them) support the mnemonic designations listed in Table 1. For example, the range [xX[:xdigit:]] matches any hexadecimal digit or the letters x or X, regardless of the machine's character set.

The *locale* refers to the character set and associated properties that apply to a given computation. The "C" locale is commonly used in the United States, where programmers are likely to process American English (or C program) text using ASCII-based computers. Other locales would be appropriate in other countries or when processing text in natural languages other than English.

Besides the lexicographical ordering of individual characters, the locale affects the choice of currency symbol; how dates and times are formatted; and the display of floating-point numbers (e.g., whether a period or a comma is used to separate the integer and

$p_1 \mid p_2$	matches either of the patterns p_1 or p_2
$p?$	matches zero or one occurrences of the pattern p
$p*$	matches zero or more occurrences of p
$p+$	matches one or more occurrences of p
$p\{k\}$	matches k occurrences of p
$p\{k,\}$	matches k or more occurrences of p
$p\{k, n\}$	matches at least k but no more than n occurrences of p

Table 2: Repetition and alternation operators in extended regular expressions.

fractional parts). In a German locale, a mnemonic identifier like [:alpha:] would match characters like ä and ß, which are not in the ASCII character set.

1.3 Extended regular expressions

Extended regular expressions are a superset of the basic regular expressions. They are used to denote repetition of a particular regular expression or a choice between regular expressions. The utilities awk and egrep use extended regular expressions. Although we will not have much use for extended regular expressions here, Table 2 summarizes them for completeness.

Extended regular expressions are *greedy*: if there is more than one possible match within a string, then the longest one is chosen. Also, as with basic regular expressions, the leftmost match is selected when more than one possibility exists.

Example 3.

- `cat|dog` matches *cat*nip and *dog*gerel.
- `abc*` matches *abccc*d instead of the other three (shorter) possibilities.

□

1.4 fgrep, grep, and egrep

Historically, there have been three versions of grep:

- fgrep ("fast grep"), which matches only substrings, not regular expressions;
- grep, which matches basic regular expressions; and
- egrep, which matches extended regular expressions.

More elaborate (and slower) algorithms are needed as the complexity of the regular expression increases. Nowadays, however, computers are so fast that the differences in memory usage and running time between the three programs are negligible in typical applications.

Some implementations, such as the open-source GNU grep, can match extended as well as regular expressions, and the three names are simply links to one common program. On other Unix systems, however, the names refer to three separate programs. For the sake of portability, it is best to invoke grep for basic regular expressions and egrep for extended regular expressions. (I have never needed fgrep.)

2 The Awk programming language

Awk is a scripting language that is particularly useful for simple text processing and data extraction tasks. A surprising number of useful jobs can be performed with a single line of Awk code. This writeup and the accompanying exercises illustrate Awk's utility.

There are several versions of Awk: the original (1978) version; the updated (1988) version, sometimes called nawk; and various GNU versions, often invoked as gawk. Depending on the particular Unix system, the command awk may refer to any one of the various versions (check the manual to determine which version is installed on your system and its command name). The discussion here refers to the version of the language described in Aho, Kernighan, and Weinberger (1988). Unless noted otherwise, however, the examples should work identically on any version of Awk.

2.1 Some one-liners

An Awk script consists of a sequence of lines of the form

pattern {*action*}

(note the curly braces around the *action*). Awk reads its input one line at a time; for each pattern that the line matches, the corresponding action is performed. Both clauses are optional: if the pattern is omitted, then every line is matched, and if the action is omitted, then the entire matched line is printed. The following examples illustrate the basic idea.

Example 1. Suppose we have an data file, say simul.dat, that contains four columns of data, such as

```
  99.0  3.14  2.17  −0.65
 100.0  3.53  1.98  −0.35
 101.0  3.99  1.76  −0.12
     ⋮
```

and so on. (Such a file might be the result of a simulation that computes and prints the values of three functions $x(t)$, $y(t)$, and $z(t)$ for selected values of t.) The command

```
awk '{print $1, $2}' simul.dat
```

outputs the first two columns:

```
 99.0  3.14
100.0  3.53
101.0  3.99
   ⋮
```

Here's what happens. Awk reads simul.dat one line at a time. Each input line is divided into *fields*, which consist of consecutive sequences of nonblank characters surrounded by whitespace. The first field is referenced by the notation $1, the tenth by $10, etc. (The special field $0 refers to the entire current line.) This script prints the first and second fields of each line, because the omitted pattern matches every line. □

Example 2. Consider again the four-column input file of Example 1. Suppose that we are interested in the norm of the computed solution, defined as $\sqrt{x^2 + y^2 + z^2}$, as a function of t. The command

```
awk '{print $1, sqrt($2^2+$3^2+$4^2)}' simul.dat
```

produces

```
 99.0  3.87182
100.0  4.06249
101.0  4.36258
   ⋮
```

The caret (^) is Awk's exponentiation operator. □

In principle, Awk can perform any desired arithmetic on the input fields. This capability makes Awk an attractive "filter" for simple data processing jobs. For instance, the output from the script in Example 2 can be used as input to a 2-d plotting program.

Table 1 shows the mathematical functions that are built into typical Awk implementations. Range errors, such as sqrt(x) for $x < 0$, either halt the script or produce infinite or not-a-number values, depending on the version of Awk.

2.2 Awk patterns

Example 3. The pattern clause can contain a mixture of regular expressions and conditional expressions. Again consider the four-column input format of the previous examples. Suppose we want a plot of $\log_{10} x$ as a function of t, where we agree to omit any data with a nonpositive values of x. A command like

```
awk '$2 > 0 {print $1, log($2)/log(10)}' simul.dat | plotprog
```

accomplishes the task, assuming that plotprog is the name of a suitable plotting program. An equivalent command (with different placement of the curly braces) is

atan2(y,x)	$\tan^{-1}(y/x)$ if $x \neq 0$ and $\pi/2 \times (\operatorname{sign} y)$ otherwise
exp(x)	e^x
log(x)	$\ln x$ (natural logarithm)
sqrt(x)	$\sqrt{x}$
cos(x)	$\cos x$
sin(x)	$\sin x$
int(x)	$\lfloor x \rfloor$ (truncate to integer)
rand()	uniformly distributed pseudorandom number in $(0,1)$
srand(x)	uses x as the seed for rand

Table 1: Built-in mathematical functions in Awk.

```
awk '{if($2 > 0) print $1, log($2)/log(10)}' simul.dat | plotprog
```

Taste and clarity dictate whether to use an explicit pattern instead of a conditional expression in an action clause. □

Regular expressions in Awk are enclosed in forward slashes: */p/* is the extended regular expression *p*. If the pattern contains a forward slash, then precede it with a backslash: /12\/25/ is the regular expression *12/25*.

Unless otherwise specified, Awk attempts to match a regular expression using the entire current input line. However, you can restrict Awk's attention to a particular field or string by using the ∼ ("matches") operator:

/abc/	true if the current line contains *abc*
\$0 ∼ /abc/	equivalent to the above
\$2 ∼ /abc/	true if the second field contains *abc*
\$2 ∼ /^abc/	true if the second field begins with *abc*
str ∼ /abc\$/	true if the variable str ends with *abc*
str !∼ /abc/	true if the variable str does *not* contain *abc*
\$2 ∼ /^abc\$/	true if the second field is precisely the string *abc*
\$2 == "abc"	another way to test for equality with *abc*
\$2 ∼ /abc\$/ && \$1 >0	true if the second field ends with *abc* and the first field is a positive number
\$2 ∼ /abc\$/ || \$1 >0	true if the second field ends with *abc* or the first field is a positive number

2.3 The patterns BEGIN and END

Two special patterns, BEGIN and END, may be used when initialization and finalization are needed. The action associated with BEGIN is performed before the first line of input is read; the action associated with END is performed after the last line of input is processed.

Example 4. The command

```
awk 'END {print NR}' file.txt
```

prints the number of lines in file.txt. In this case, there are no patterns to match, so the script outputs nothing as it reads its input—but Awk silently increments NR after each input line is read successfully. When the end of the input is reached, the current line count is printed. This script handles empty files correctly: it prints 0, because no line can be read successfully. □

Empty actions are permitted. For instance, a statement like BEGIN {} is accepted but unnecessary. The BEGIN clause can simply be omitted if no special initialization is required. Similar comments apply to END.

Example 5. The geometric mean of n positive values is the nth root of their product. Suppose the input consists of at most one positive value per line (blank lines are permitted). The following Awk script computes and prints the geometric mean of the data:

```
BEGIN    {prod = 1}
NF > 0   {prod *= $1; n++}
END      {if(n > 0) print prod^(1/n)}
```

The running product must be initialized to 1 before the data is processed, so the BEGIN pattern is required. NF is a built-in variable that Awk sets equal to the Number of Fields in the current line. Hence, only nonblank lines match the pattern NF > 0, in which case we simply update the running product. The variable n counts the number of nonblank lines, which under our assumptions is the number of values in the input. When the end the input is reached, Awk computes the nth root of the data; if the input is empty or all blank, then the script prints nothing. Because uninitialized variables are either zero or the null string, depending on context, it is not necessary to include the initialization n = 0 in the BEGIN clause, although there is no harm in doing so. □

2.4 Awk variables

Variables in Awk are not declared; they simply spring into existence as soon as they are referenced. Uninitialized Awk variables are set either to zero or to the null string as the context requires. All arithmetic in Awk is done in double precision. In Awk, there is no distinction between 1/n and 1.0/n; each gives a double-precision approximation to $1/n$.

In fact, *all Awk variables are strings* (just as in the shell). When a variable is used in an arithmetic context (e.g., x+y), its contents are converted to a numeric value, the appropriate arithmetic operation is performed, and the results are converted back to an appropriate string of digits.

Example 6. The contents of two Awk variables can be concatenated simply by writing them next to each other. The sequences

x = 1; y = 2; print x y   and   x = "1"; y = "2"; print x y are equivalent, and both print 12. The statements

```
print x, y
print x " " y
```

are equivalent, and both insert a space between x and y; the first is preferable. □

Example 7. If the contents of a variable are not numeric, then Awk silently substitutes zero in an arithmetic context. To check whether a variable contains a valid digit string, add 0 to it. The following command prints all the lines in input.txt whose first field can be interpreted as a number:

```
awk '$1 + 0 == $1' input.txt
```

Awk supports scientific as well as the usual fixed-point notation. For example, 1000, 1.0e+03, and 1e3 are all equivalent. □

2.5 The variables $0, NF, NR, and FS

Awk maintains several built-in variables, including:

- $0, the entire current line;
- NF, the Number of Fields in the current input line;
- NR, the Number of the current Record. Each line forms one record, and the first record is numbered 1;
- FS, the current Field Separator (the default is any collection of blanks and tabs).

The first three of these variables are initialized automatically when each input line is read. FS can be set by the user, usually in a BEGIN clause, to process files whose fields are separated by characters other than whitespace.

Example 8. Data from spreadsheet programs sometimes can be saved as plain text files, with the fields separated by tabs. Such files can be processed by prefacing the script with

```
BEGIN {FS = "\t"}
```

Files with comma-separated values often are denoted with the suffix .csv. They can be processed with

```
BEGIN {FS = ","}
```

Alternatively, the field separator can be set (in this case to a comma) using the –F command-line option:

```
awk –f prog.awk –F , input.csv
```

Always use the special notation \t to indicate tab field separators, because literal tab characters are difficult to distinguish from ordinary spaces in printouts and most text editors. □

The syntax of the statements within each action clause is very similar to C: assignment, for, and while statements have the same format in Awk as in C. All four basic arithmetic operations ($+$, $-$, $*$, $/$) are provided, along with exponentiation (^) and the incrementation operators (+=, −−, etc.) There's also a printf statement with the same syntax and formatting directives as the standard C library function.

The end of an Awk statement may be indicated with a semicolon, as in C, or simply by a line break, as in Fortran. A semicolon is required if more than one statement appears on one line.

Example 9. The following command prints the contents of input.txt with line numbers:

```
awk '{printf("%4d %s\n", NR, $0}' input.txt
```

The line numbers are right-justified in a field that is 4 digits wide. □

Example 10. The following command prints the sum of all values on each input line:

```
awk '{sum=0; for(j=1; j<=NF; j++) sum += $j; print sum}' input.txt
```

The notation $j refers to the *j*th field. Because there are no curly braces to denote otherwise, the body of the for loop is sum += $j. (The += operator is borrowed from C; the statement is equivalent to sum = sum + $j.) A semicolon is required after the loop body because another statement, print, appears on the same line. (A semicolon is also required after the initialization of sum for the same reason.) □

The individual fields of the current input line can be referenced by any nonnegative integer-valued expression. For instance, $(k+1) refers to the $(k+1)$st field.

Example 11. Awk's print statement formats numeric values with six decimal digits of precision; trailing zeros are suppressed and scientific notation is used only when necessary. Fancier formatting requires printf instead. Consider again the 4-column output file at the beginning of this section; the command

```
awk '{printf("%6.2f %.9e\n", $1, sqrt($2^2+$3^2+$4^2)}' simul.dat
```

produces

```
 99.00  3.871821277e+00
100.00  4.062486923e+00
101.00  4.362579512e+00
```

and so on. □

2.6 Command-line format

An Awk script may be stored in a file, say prog.awk. The script is invoked with a command line like

```
awk –f prog.awk input.dat
```

which reads the data from the file input.dat.

If no file is specified, Awk reads from the standard input. When multiple files are specified, Awk processes them in the order of their appearance on the command line. For example,

```
awk –f prog.awk file1.dat file2.dat
```

processes file1.dat first, followed by file2.dat.

The names of the script and input files are immaterial to Awk. Although we have suffixed the script file with .awk and the input files with .dat for illustration, these suffixes are not necessary; any convenient naming scheme will do.

2.7 The structure of Awk scripts

Awk scripts tend to describe *what* you want to compute, not *how* to compute it. Every script forms an implicit loop that begins with the first line of input and automatically terminates when no more input is available. In other words, an Awk script of the form

$$\begin{array}{ll} \mathit{pattern}_1 & \{\mathit{action}_1\} \\ \mathit{pattern}_2 & \{\mathit{action}_2\} \\ \vdots & \\ \mathit{pattern}_n & \{\mathit{action}_n\} \end{array}$$

is equivalent to the following pseudocode:

```
do while(input is available)
    read(line)
    if(line matches $pattern_1$) perform $action_1$
    if(line matches $pattern_2$) perform $action_2$
    ⋮
    if(line matches $pattern_n$) perform $action_n$
enddo
```

Each input line is compared to the patterns in the order in which they appear in the script. Every time there's a match, Awk performs the corresponding action. A line that matches no pattern is silently discarded. Awk handles all the details of checking for end-of-file, splitting each input line into fields, matching regular expressions, and so on.

Example 12. Suppose that prog.awk is

```
 /Tomasz/   {print 6}
 /Tomas/    {print 5}
 /Tom/      {print 3}
```

Then the command

```
echo Tom Hanks | awk –f prog.awk
```

prints 3, because the single input line matches only the last pattern. The command

```
echo Tomasz Stanko | awk –f prog.awk
```

prints

```
6
5
3
```

because Tomasz, Tomas, and Tom all match substrings of Tomasz Stanko. □

No particular formatting of pattern-action pairs is required. They can be placed on one line like this:

$pattern_1$ {$action_1$} $pattern_2$ {$action_2$}

or spread among several lines, like this:

```
pattern₁ {
        action₁
}
pattern₂ {
        action₂
}
```

However, a pattern must not be split from the opening brace of its corresponding action clause by a line break, because that changes the meaning of the program. The script

```
pattern
{action}
```

prints all lines that match *pattern* and performs *action* for every input line.

The next statement causes Awk to discard the current line and move on to the next. It allows you to stop the matching process as soon as a given pattern is matched. For example, an Awk script of the form

```
 pattern₁   {action₁; next}
 pattern₂   {action₂; next}
            {action₃}
```

is equivalent to the loop

```
do while(input is available)
    read(line)
    if(line matches $pattern_1$) then
        perform $action_1$
    else if(line matches $pattern_2$) then
        perform $action_2$
    else
        perform $action_3$
    endif
enddo
```

Example 13. Let us modify the script in Example 12 as follows:

```
/Tomasz/   {print 6; next}
/Tomas/    {print 5; next}
/Tom/      {print 3; next}
```

Now the command

```
echo Tomasz Stanko | awk –f prog.awk
```

prints only 6. Once Tomasz is matched, the next statement in the corresponding action causes Awk to discard the input line without checking the remaining patterns. □

Table 2 summarizes the remaining built-in functions in Awk. Most of these are self-explanatory. The next example shows how to use split, which is slightly more complicated.

Example 14. Consider a roster of players on a football team that lists them as last name, first name, position, height, and weight:

```
Aikman, Troy   QB    6'04"   220
Namath, Joe    QB    6'2"    200
Tillman, Pat   OLB   5'11"   202
```

How can we use Awk to find the tallest player? A lexicographical comparison fails because some heights contain leading zeros in the inches field and others don't. We need to convert each player's height into inches (or other scalar unit); split makes this easy:

```
{      split($4, ht, /['"]/)
       inches = ht[1]*12 + ht[2]
       if(inches > tallest) {
           tallest = inches
           player = $0
       }
}
END    {print player}
```

index(s, x)	returns the position in string s where the substring x occurs, or 0 otherwise. The first character of s is position 1.
length(s)	the length of string s; if s is omitted, then the length of the current line
match(s, p)	returns the position in string s where the regular expression p occurs, or 0 otherwise. The built-in variables RSTART and RLENGTH, respectively, are set to the position and length of the matched string.
sub(p, r, s)	substitutes r for the first occurrence of regular expression p in s; returns the number of substitutions
sub(p, r)	substitutes r for the first occurrence of regular expression p in the current line ($0); returns the number of substitutions
gsub(p, r, s)	like sub, except that *all* occurrences of p are replaced with r; returns the number of substitutions
split(s, a, p)	splits string s into subfields using the characters in p as the field separators. Places the first subfield into a[1], the second into a[2], etc. If p is empty, then s is split into individual characters
sprintf($s, f, e_1, e_2, \ldots$)	like the standard C library function sprintf: expressions e_1, e_2, ... are formatted according to f and the result placed in s
substr(s, k, ℓ)	the substring of s of length ℓ beginning at position m, counting from 1
system(s)	runs a shell to execute the command s; returns the exit status
tolower(s)	returns string s with uppercase letters replaced with lowercase
toupper(s)	returns string s with lowercase letters replaced with uppercase

Table 2: Other built-in functions in Awk.

This script can be placed into a file, say tallest.awk, and invoked with a command like

```
awk –f tallest.awk players.dat
```

A separate file for the Awk script is desirable here because the script is considerably longer than a "one liner" and because the single and double quotation marks are significant to the shell.

Each player's height is the fourth field. (The names occupy two fields, not one!) The function split partitions $4 into two subfields, namely feet and inches, and places them into ht[1] and ht[2], respectively (space for the two-element array ht is allocated dynamically). The subfields are separated by either a single or a double quotation mark; hence the use of a regular expression range in square brackets to indicate either one of the two. (If we included only the single quote as a separator, then the double quote would be appended to the digits in ht[2]; the resulting string would not be a valid numeric value).

The variable tallest holds the height of the tallest player found so far (it's automatically initialized to 0, so we don't need a BEGIN block). The variable player holds the unedited information corresponding to the current tallest player. This script, then, prints the first line of this three-man roster. □

2.8 Awk arrays

Awk provides a one-dimensional array capability. Like other variables in Awk, arrays require no declaration; space for them is allocated when they are first mentioned. The contents of each element is a string, which, upon first use, is empty or zero depending on context. For example, if the input consists of one value per line, then the following code fragment illustrates how to read in all of the input data into an array before processing it:

```
        {x[NR] = $1}
END     {for(j = 1; j <= NR; j++)} process x[j]}
```

In this particular case, it is natural to start the loop counter at 1, insofar as the first input record (line) is numbered 1. Zero-based indexing is possible simply by adjusting the subscript when reading:

```
        {x[NR–1] = $1}
END     {for(j = 0; j < NR; j++)} process x[j]}
```

In typical programming languages, array subscripts are integers, as in the code above. However, because all Awk variables are strings, Awk array indices, are too. Consequently, you may use any string as a subscript. In other words, Awk arrays are *associative*[1], insofar as they associate a *key* (the string that comprises the subscript) to a value (the array element indexed by the key). Awk handles all the details of memory management and table lookup.

[1]Associative arrays are also known as *dictionaries* or *hash tables*.

Example 15. x[1] and x["1"] are synonymous. □

Example 16. One way to count the number of players at each position in the football roster in Example 14 is to use the position abbreviation as a subscript. Whenever a new subscript appears, the corresponding array element is created and initialized to 0. The positions are the third field, so we simply need to increment the count each time. The for statement at the end loops through all the array keys:

```
          {count[$3]++}
    END   {for(position in count) print count[position]}
```

□

Example 17. Suppose we have a simple list of inventory like this:

```
hammer  2
nail  100
wrench  3
hammer  1
nail  50
```

Awk can tally the number of each item as follows:

```
          {count[$1] += $2}
    END   {for(item in count) print item, count[item]}
```

The result is:

```
hammer  3
wrench  3
nail  150
```

As illustrated above, Awk provides a special form of the for construct to iterate through all the keys in a given array. This construct is essential here, because we cannot know the name of each item in inventory until all the input has been read. The for loop terminates when the set of keys is exhausted.

The order in which each item is printed may differ from that shown here, depending on how each Awk implementation manages its arrays. Pipe the output to sort to assure an alphabetized listing. □

2.9 Putting it all together

Tools like Awk are especially helpful for scanning logfiles from simulation programs, provided that care is taken to make them easy to parse by machine. Often, however, little thought is put into the formatting of logfiles. It's common in practice to see output like

```
Starting simulation. Input: x0=1., –9., 0.
Parameter c= 2.14. ExtraCheckFlag is off.
At time t=0.10000 solution is x= 1.1456  y= –9.2300
z= 0.0
Error = 0.0009
At time t=0.20000 solution is x= 1.2317  y=–10.2911
z= 0.0000
Error = 0.2161
⋮
```

and so on for perhaps thousands of lines.

Such a format may be fine for initial debugging, but once the program is working, other questions become important, and the limitations of this formatting become apparent. Suppose, for instance, that the error begins to grow unexpectedly as the simulation runs. The following questions might arise: Is there a programming error or a typographical error in the input data? Is the numerical method unstable? Or is there an intrinsic problem with the mathematical model? One of the joys of scientific computing is that it's possible to spend anywhere from a couple of hours to several months answering these questions, depending on the complexity of the simulation.

Tools like Awk can be useful for probing the output to help answer these questions—but the job is much easier if the output is formatted with machine parsing in mind. In the output above, for instance, the values of the solution components *x*, *y*, and *z* are separated by an equals sign with varying amounts of whitespace; the number of significant digits in the error is far from constant; and the solution components appear on multiple lines.

While it's possible to write an Awk script to parse this type of output, a bit of work is involved. If changes to the original program's source code are permitted, however, then one can add a couple of extra print statements, and make minor changes to the existing ones, to transform the output into something like this:

```
# Starting simulation. Code version 02/11/05.
# Begin input
x0   1.000  2.000 –9.000
c   2.14
ExtraCheckFlag  off
# Begin simulation
# Output lines have the form t (time) x,y,z (solution components)
# Error lines are the 2-norm of the estimated error
t 0.10000   1.1456  –9.0010   0
```

```
Error  9.1434e–04
t 0.20000   1.2317  –10.2911   1.012e–05
Error 2.1611e–02
⋮
# End simulation
```

Notice two key features:

- “Comment” lines begin with #
- All other lines start with a key word.

The # convention is useful because these lines are easily skipped in Awk and other programs:

- Place the pattern-action pair

  ```
  /^#/ {next}
  ```

  first in an Awk script; the comment lines are silently discarded.

- A command line of the form

  ```
  grep –v '^#' logfile
  ```

  copies all the non-comment lines in logfile to the standard output.

The keyword convention is useful because parameters like x0 can be scanned quickly. If x0 denotes the initial conditions, then a command like

```
grep '^x0' *.log
```

lists the initial conditions that are used in every log file in the current directory.

Suppose that we want a plot of error versus time. The following Awk script sends (time, error) pairs to the standard output (where they can be piped to a suitable plotting program):

```
/^t /       {t = $2}
/^Error/    {print t, $2}
```

The pattern-action pairs must appear in this order because one Error line appears after each t line in the log file. When Awk encounters a line starting with t, the current value of the time is silently remembered. Nothing is printed until an Error line is encountered, and the structure of the log file ensures that the printed value of t corresponds to the value of the error. If the script is saved as the file t_error.awk, then a command of the form

```
awk –f t_error.awk file.log | plotprog
```

generates the plot (assuming that plotprog is the name of a suitable plotting program).

Finally, suppose that we have a collection of log files in the current directory, each suffixed .log. For each log file, we want to know the earliest time *t* at which the error exceeds 1.5. We can use an Awk and a shell script to accomplish the task.

The Awk script is a modification of the previous one:

```
/^t /                   {t = $2}
/^Error/ && $2 > 1.5    {print t; exit}
END                     {print "never"}
```

The second pattern matches lines that begin with Error and whose second field is larger than 1.5. The exit statement causes Awk to halt (so at most one time *t* is printed). On the other hand, if the error never becomes larger than 1.5, then it might be helpful to print a message to this effect; the END clause handles this case.

We can save this script as bigerror.awk and combine it with a shell script to loop over all the log files in the current directory, as follows:

```
for file in *.log
do
   echo file $(awk –f bigerror.awk $file)
done
```

In this way, we obtain one $(\text{name}, \text{time})$ pair for each log file.

References

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