

AWK PROGRAMMING

Student Workbook

AWK PROGRAMMING

Jim McNally

Published by ITCourseware, LLC, 7245 South Havana Street, Suite 100, Centennial, CO 80112

Editor: Rick Sussenbach

Copyright © 2011 by ITCourseware, LLC. All rights reserved. No part of this book may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording, or by an information storage retrieval system, without permission in writing from the publisher. Inquiries should be addressed to ITCourseware, LLC, 7245 South Havana Street, Suite 100, Centennial, CO, 80112. (303) 302-5280.

All brand names, product names, trademarks, and registered trademarks are the property of their respective owners.

CONTENTS

Chapter 1 - Course Introduction	7
Course Objectives	8
Course Overview	10
Using the Workbook	11
Suggested References	12
Chapter 2 - Introduction to awk	15
Introduction to awk	16
How awk Programs Work	18
Running awk Programs	20
Examples	22
More Examples	24
Even More Examples	26
Labs	28
Chapter 3 - Patterns	31
Summary of Patterns	32
BEGIN and END	34
Expressions	36
String-Matching Patterns	38
Extended REs in awk	40
Range Patterns	42
Labs	44
Chapter 4 - Actions	47
awk Actions	48
Expressions	50
Operators	52
Flow Control	54
More Flow Control	56
next, break, continue, and exit Statements	58
Built-in Variables	60
Labs	62

Chapter 5 - Input and Output	65
Formatted Output with printf	66
Output Into Files	68
Output Into Pipes	70
Input Separators	72
Multiline Records	74
The getline Functions	76
More getline	78
Passing Parameters Into a Script	80
Labs	82
Chapter 6 - Functions	85
Built-In Arithmetic Functions	86
Built-In String Functions	88
More String Functions	90
User Defined Functions	92
Local Variables	94
Labs	96
Chapter 7 - Arrays	99
Arrays	100
Associative Arrays	102
The Array for Statement	104
The Array in Operator	106
Deleting Array Elements	108
The Split Function	110
Multidimensional Arrays	112
Command Line Parameter Passing	114
Labs	116
Appendix - Regular Expressions	119
Introduction	120
What is a Regular Expression?	122
Literal Regular Expressions	124
Regular Expressions: ^, \$, \	126
Regular Expressions: ., [s]	128

More about [s]	130
Regular Expressions: *	132
More about *	134
Regular Expressions: {m,n}	136
Labs	138
 Solutions	143
 Index	155

CHAPTER 1 - COURSE INTRODUCTION

COURSE OBJECTIVES

- * Write applications using the awk programming language.

COURSE OVERVIEW

- * **Audience:** This is a programming course designed for UNIX system application developers, administrators, and advanced users. You will write many programs in this class.
- * **Prerequisites:** The ability to write programs in a high-level language (such as C or shell) is very helpful in completing the lab exercises and understanding the lectures. A good working knowledge of the UNIX environment is necessary.
- * **Classroom Environment:**
 - Individual terminals.
 - Reference materials.

USING THE WORKBOOK

This workbook design is based on a page-pair, consisting of a Topic page and a Support page. When you lay the workbook open flat, the Topic page is on the left and the Support page is on the right. The Topic page contains the points to be discussed in class. The Support page has code examples, diagrams, screen shots and additional information. **Hands On** sections provide opportunities for practical application of key concepts. **Try It** and **Investigate** sections help direct individual discovery.

In addition, there is an index for quick look-up. Printed lab solutions are in the back of the book as well as on-line if you need a little help.

The Topic page provides the main topics for classroom discussion.

The Support page has additional information, examples and suggestions.

JAVA SERVLETS

THE SERVLET LIFE CYCLE

- * The servlet container controls the life cycle of the servlet.
- When the first request is received, the container loads the servlet class.

Topics are organized into first (★), second (➤) and third (■) level points.

- As with Java's `finalize()` method, don't count on this being called.

- * Override one of the `init()` methods for one-time initializations, instead of using a constructor.

- The simplest form takes no parameters.

```
public void init() { ... }
```

- If you need to know container-specific configuration information, use the other version.

```
public void init(ServletConfig config) { ... }
```

- Whenever you use the `ServletConfig` approach, always call the superclass method, which performs additional initializations.

```
super.init(config);
```

Pages are numbered sequentially throughout the book, making lookup easy.

CHAPTER 2

SERVLET BASICS

Hands On:

Add an `init()` method to your `Today` servlet that initializes along with the current date:

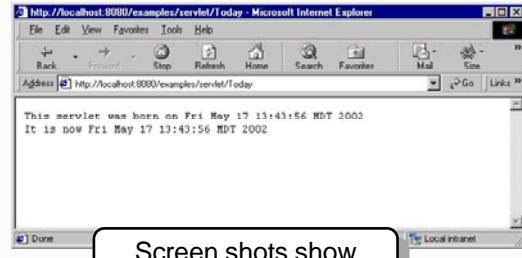
`Today.java`

```
...
public class Today extends GenericServlet {
    private Date bornOn;
    public void service(ServletRequest request,
                        ServletResponse response) throws ServletException, IOException
    {
        ...
    }
}
```

Code examples are in a fixed font and shaded. The on-line file name is listed above the shaded area.

Callout boxes point out important parts of the example code.

The `init()` method is called when the servlet is loaded into the container.



Screen shots show examples of what you should see in class.

SUGGESTED REFERENCES

Aho, Alfred V., Brian W. Kernighan, and Peter J. Weinberger. 1988. *The AWK Programming Language*. Addison-Wesley. ISBN 020107981X.

Dougherty, Dale and Arnold Robbins. 1997. *sed & awk, 2nd Edition*. O'Reilly & Associates, Sebastopol, CA. ISBN 1565922255.

Gilly, Daniel and Arnold Robbins. 1999. *UNIX in a Nutshell, 3rd Edition*. O'Reilly & Associates, Sebastopol, CA. ISBN 1565924274.

Powers, Shelley, Jerry Peek, Tim O'Reilly, and Mike Loukides. 2002. *UNIX Power Tools, 3rd Edition*. O'Reilly & Associates, Sebastopol, CA. ISBN 0596003307.

Robbins, Arnold. 2001. *Effective awk Programming, 3rd Edition*. O'Reilly & Associates, Sebastopol, CA. ISBN 0596000707.

CHAPTER 2 - INTRODUCTION TO AWK

OBJECTIVES

- * Explain what awk is and where it is useful.
- * Invoke awk from the command line and from awk scripts.
- * Use awk to extract data from files in several ways.

INTRODUCTION TO AWK

- ※ *awk* is a programming language quite fit for a wide variety of data manipulation and computational tasks.
- ※ With awk programs you can:
 - Change the format of data.
 - Check data validity.
 - Select, manipulate, and display data that meet a condition or set of conditions.
 - Print reports based on file or input data.
 - Perform calculations on numeric data.
- awk action code looks a lot like C.
- **awk** is a UNIX utility that interprets and runs awk programs, or scripts, that you write.

There are several versions of awk, including *nawk* (new awk) which has many additional features the original awk did not support. On many UNIX systems, **awk** is actually a link to **nawk**, and all users actually run **nawk** even when they think they're running **awk**. This usually works fine since nawk is a superset of awk.

If you try to run an awk (or nawk) script and it doesn't work right or you get a syntax error, try running it with **nawk** explicitly. If this fixes the problem, just use **nawk** until you can talk to the system administrator about possibly fixing the problem with a link or alias.

HOW AWK PROGRAMS WORK

- ※ **awk** reads input from a file or from standard in, one line (record) at a time.
- ※ If the current line is matched by one or more specified patterns, then the action associated with each matched pattern is performed.
- ※ Patterns can be expressions, REs, ranges, or special patterns.
- ※ Actions include operations, functions, control flow, output.
- ※ In an awk program, actions associated with a pattern are enclosed in braces.
 - The general format of an awk program is:

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

- ※ awk is a UNIX filter: it is designed to process a stream of data which it reads from standard in and writes to standard out.

Looking at the points on the facing page, you can see that the first two basically do what sed does. The next one is similar to sed, but sed can't deal with expressions (like "do this when the value of the fifth field is equal to 5000"). The next two include capabilities which sed cannot hope to perform. These are the things which make awk so useful and powerful, yet easy to use.

RUNNING AWK PROGRAMS

- ※ awk reads records from the input.
- ※ When a record is read in, it is automatically split into fields, which are variables referenced as **\$1, \$2, \$3, . . .**
- ※ awk programs can appear on the shell command line, or they can be put in a file.
- ※ Examples of command line programs:
 - When no pattern is specified for an action, the action is performed on every line.

```
awk '{print $2}' cars
```

- When no action is specified for a pattern, the default action is to print the lines matching the pattern.

```
awk '/ford/' cars
```

- Pattern: Match all lines containing **ford**.
Action: Print the second field.

```
awk '/ford/ { print $2 }' cars
```

You can think of sed and grep as being subsets of awk as far as functionality is concerned. In fact, the second example on the topic page:

```
awk '/ford/' cars
```

does exactly what

```
grep ford cars
```

does.

EXAMPLES

1. Show information on cars older than 1981:

```
awk '$3 <= 80 { print "1980 or older: " $1, $2 }' cars
```

2. Similar example but put program in a file:

```
awk -f 80older.awk cars
```

➤ where *80older.awk* contains:

```
BEGIN { print "1980 or older"
        print "_____"
    }
$3 <= 80 { print $1,$2,"Year: ",$3,"Price: ",$5}
```

3. The number of fields in the line is stored by awk in the built-in variable **NF**:

```
awk '{print NF, $1, $2, $NF}' cars
```

4. The number of lines read so far (the current record number) is stored in **NR**:

```
awk '/^ford/ { print NR, $0 }' cars
```

➤ Note: **\$0** refers to the entire line.

5. Display the number of lines in the file:

```
awk 'END { print NR }' Memos/orals
```

Another built-in variable that can be useful is **\$0**. As you might have guessed, **\$0** is automatically loaded with the entire line as it is read in. This can be used to add additional things to existing intact lines without modifying the original line.

MORE EXAMPLES

1. List expensive cars, sorted by high price:

```
awk '$5 > 8000' cars | sort +4 -nr
```

2. Foreign (to the USA) cars:

```
awk -f foreign.awk cars
```

➤ Where *foreign.awk* contains:

```
$1 == "toyota" { print $3, $1, $2, " : $", $5 }
$1 == "fiat"   { print $3, $1, $2, " : $", $5 }
$1 == "honda"  { print $3, $1, $2, " : $", $5 }
```

➤ An equivalent *foreign.awk*:

```
$1 == "toyota" || $1 == "fiat" || $1 == "honda"
{ print $3, $1, $2, " : $", $5 }
```

3. Program for total value on the car lot:

```
{ total = total + $5 }
END { printf("Total inventory value: %10.2f\n", total)
}
```

Another thing to note is the use of the comparison operator `==`. Although some languages use `=` both to assign a value and to make comparisons, awk follows C syntax and differentiates between them. To set some variable equal to a value, use the assignment operator `=` as in:

```
variable1 = 123
```

To test for equality (even with a string) use the comparison operator `==` as in:

```
$1 == "toyota"
```

The bottom example utilizes the `printf()` function. If you're not familiar with `printf()` in C, we will cover it in detail later.

EVEN MORE EXAMPLES

1. Print the third line:

```
awk 'NR == 3' anyfile
```

2. Print lines in which the last field is greater than 5000:

```
awk '$NF>5000' cars
```

3. Print total number of lines containing **chevy**:

chevycount.awk

```
/chevy/ { n = n + 1 }
END      { print n }
```

4. Print all lines longer than 66:

```
awk 'length($0) > 66' orals
awk 'length()    > 66' orals
awk 'length      > 66' orals
```

Note: **length()** is a built-in function.

Note: These three examples are equivalent.

5. Print sorted list of user ids on the system:

```
cat /etc/passwd|awk -F: '{print $3, $1}' |sort -n
```

Note: The **-F** option sets the field separator.

Notice that an equivalent sed expression to the first example would be

```
sed -n 3p anyfile
```

Also note in example 5 the use of the comma in the print statement. It tells awk to separate the two variables with a white space (default) between them. If you don't use the comma, the two variables will be printed directly adjacent to one another. If you want something different between them, you can explicitly insert it as in:

```
{print $3 " " $1}
```

which will put three spaces between fields three and one, or you can use **printf()**.

The default white space happens because a white space is the default value for the special variable **OFS** (Output Field Separator). If you wanted the comma to insert a different character (or string), you need only reset **OFS** to the desired string, as in:

```
OFS = ":"
```

now, **{print \$3,\$1}** will print the third field followed by a colon followed by the first field:

```
0:root
1:daemon etc.
```

LABS

- 1** Use awk to swap the first and last names on each line of the *list* file, and insert a comma immediately after the last name. Thus,

Dino Flintstone 94111

would become

Flintstone, Dino 94111

- 2** Write an awk program that calculates and prints the total values of the cars on the lot, broken out by manufacturer. The output should look like:

```
Ford: 47000  
Chevy: 8050  
Plym: 2500  
Volvo: 9850  
Fiat: 450  
Honda: 6000
```

- 3** Enhance the program in the previous exercise to print a table that displays in the following format:

<u>Make</u>	<u>Total Value</u>	<u>Average Value</u>	<u>Average Age</u>
Ford	47000	11750	13
Chevy	8050	2683	17
Plym	2500	2500	15
Volvo	9850	9850	14
Fiat	450	450	27
Honda	6000	6000	11

CHAPTER 6 - FUNCTIONS

OBJECTIVES

- * Use awk's built-in string and arithmetic functions.
- * Define your own functions.
- * Create local variables in functions.

BUILT-IN ARITHMETIC FUNCTIONS

- ※ awk's arithmetic operators (+, -, *, /, etc.) do a limited set of algebraic math.
- ※ More advanced math is done with awk's arithmetic functions.
- ※ Functions return values, based on their arguments and what the function does.
 - Return values can be captured to variables using the assignment operator:

```
x = cos(y)
```

- In context, functions can return values right on the line:

```
print cos(y)
```

- Functions can be nested:

```
print cos(log (rand()))
```

- ※ Built-in functions take zero, one or two arguments: syntax errors will result from improper numbers of arguments.

Built-in Arithmetic Functions

Function	Value returned
atan2(y,x)	arctangent of y/x (range $-\pi$ to π)
cos(x)	cosine of x (x in radians)
exp(x)	e^x (exponential function of x)
int(x)	integer portion of x
log(x)	base e logarithm of x
rand()	random number between 0 and 1
sin(x)	sine of x (x in radians)
sqrt(x)	square root of x
srand(x)	sets x as seed value for rand()

rand() and **srand(x)** are interesting functions.

rand() returns a pseudorandom sequence of numbers, one value for each time it is called. **srand()** sets a seed value for the pseudorandom algorithm used by **rand()**.

If **rand()** is called without first calling **srand(x)**, then a default value for the seed is used, and the sequence, while itself close to random, will be the same each time the program is run. If **srand(x)** is called with x set to an integer, the sequence will be different for different values of x . If **srand()** is called without a value, a seed will be generated based on the time of day (more nearly really random).

BUILT-IN STRING FUNCTIONS

- ※ awk has a group of built-in functions designed to work with strings.
- ※ String functions are divided into several groups:
 - Functions which return information about a string: **length()**, **index()**, **match()**.
 - Functions which return a subset of the initial string as defined by arguments: **substr()**.
 - Functions which change the target string according to a set of rules: **gsub()**, **split()**, **sprintf()**, **sub()**.
- ※ Regarding functions which return information:
 - **length(s)** returns the number of characters currently stored in variable **s**:

```
string = "abcdefg"
length(string)
```
 - **index(s,t)** returns the position (in characters) of the string **t** in the string **s** or **0** if not found.

```
string = "abcdefg"
print index(string, "cd")
```
 - **match(s,t)** returns the character position of string **t** in string **s**, or **0** if not found.
 - This behaves similarly to **index()** except it sets **RSTART** and **RLENGTH** variables.

Built-In String Functions

Function	Description
gsub(r,s)	Substitute s for r globally in \$0 , return number of substitutions made.
gsub(r,s,t)	Substitute s for r globally in string t , return number of substitutions made.
index(s,t)	Return first position of string t in s , or 0 if t is not present.
length(s)	Return number of characters in s .
match(s,r)	Test whether s contains a substring matched by r ; return index or 0 ; sets RSTART and RLENGTH .
split(s,a)	Split s into array a on FS , return number of fields.
split(s,a,fs)	Split s into array a on field separator fs , return number of fields.
sprintf(fmt,list)	Return list formatted according to format string fmt .
sub(r,s)	Substitute s for the leftmost longest substring of \$0 matched by r ; return number of substitutions made.
sub(r,s,t)	Substitute s for the leftmost longest substring of t matched by r ; return number of substitutions made.
substr(s,p)	Return suffix of s starting at position p .
substr(s,p,n)	Return substring of s of length n starting at position p .

MORE STRING FUNCTIONS

- * Functions which return a subset:

- **substr(s,p)** returns a substring of string **s** starting at position **p** to end of string.
- **substr(s,p,n)** returns a substring of string **s** starting at position **p** of length **n**.

```
string = "abcdefg"  
print substr (string, 4)  
print substr (string, 4, 2)
```

This will print **defg**.

This will print **de**.

- * Functions which change a string:

- **gsub(r,s)**, **gsub(r,s,t)** will substitute the string **s** for the string **r** globally in **\$0** (**gsub(r,s)**) or in the string **t** (**gsub(r,s,t)**).

```
string = "abcdefgabcdefg"  
gsub (/abc/, "xyz", string)  
print string
```

This will print
xyzdefgxyzdefg.

- **sub(r,s)**, **sub(r,s,t)** works the same as **gsub()** except it only changes the leftmost substring found.

```
string = "abcdefgabcdefg"  
sub (/abc/, "##", string)  
print string
```

This will print
##defgabcdefg.

- **sprintf (format, expression list)** will return a formatted string based on its argument list (just like **printf()**).

gsub() and **sub()** allow awk scripts to do very complex editing functions. For example,

```
{gsub (/214-/,"213-");print}
```

will replace all other codes of **214-** in a list of phone numbers with **213-**. There can be a problem, however; what if the phone number is **214-214-1234**? In this case, **gsub()** will change them both to **213-**. You can fix this by using **sub()** instead, which will only change the first **214-**:

```
{sub (/214-/,"213-"); print}
```

When using **sub(r,s,t)** or **gsub(r,s,t)**, the special character **&** can be used in the replacement string **s**. It will stand for (and be replaced by) the pattern matched by **r**, which allows you to use RE's to match a pattern, and use the pattern found as part of the replacement string.

```
{sub (/ [0-9] [0-9] [0-9] /,"(&)",number); print}
```

will put parentheses around the area code in a list of phone numbers.

USER DEFINED FUNCTIONS

- * nawk allows the user to define functions.

```
function name (parameter list) {  
    statement  
    :  
}
```

- * The statements are standard awk expressions and statements.
- * The parameter list is a sequence of variable names separated by commas.
 - The variables are then used within the body of the function.
- * A function can return a value with the **return** statement.
- * awk functions can be defined anywhere a pattern action statement is legal.

```
print fred(1,2,"abc")  
function fred (a,b,c)  
{  
    print c,b,a  
    return a  
}
```

- This will print:

```
abc21  
1
```

Some things to remember about awk functions:

- There cannot be any spaces between the name of the function and the opening parenthesis when you call a user-defined function.
- In the function definition, newlines are not necessary following the opening curly brace or before the closing one.
- If a function has a **return** statement, the value following **return** is optional. If there is no value provided, the actual value returned is undefined: that is, running the same script on different machines or UNIXs may give different results. Functions can be recursive —that is, they can call themselves (but be careful!!).
- The functions only exist for as long as the script is running. You cannot define a function in an awk script and have a second script then call the function (unless you put a copy of the function definition in the second script too).

LOCAL VARIABLES

- * Variables inside functions can behave in one of two ways:
 - Variables declared outside the function or declared in the body of the function are global throughout the entire script.

```
function printline() {  
    print NR,$,0  
}
```

- Variables declared inside the parentheses as arguments fed into the function are local to the function.

```
function fred (a,b,c)  
{  
    print c,b,a  
    a= "fred"  
}  
a="x"; b=3; c=0
```

fred(a,b,c) → This will print **03x**.

fred("one",2,3) → This will print **32one**.

print a → This will print **x**.

Local variables are very important, especially if you (or others) begin reusing functions in other awk scripts. Making variables local to a function ensures that there will be no overlap between function variables and other variables of the same name. If you need to make other function variables local (ones that aren't going to be passed in from the outside), just put them at the end of the list in the function declaration.

```
function foo (a,b,c,x,y,z) {  
    print a,b,c  
    x=a; y=b; z=c  
    print x,y,z  
}  
foo(1,2,3)
```

Even though only 3 arguments are passed in, all 6 will be local variables.

When a variable is passed in to a function, a copy of the variable is made. If the function changes the value of the local variable, it doesn't change the original value outside the function. This works differently with arrays, as we will see in the next chapter.

LABS

- 1** Write a program which accepts a number from standard in and prints it, followed by the square root of the number.
(Solution: *square.awk*)

- 2** Expand the above to accept strings also, but to recognize them as strings and print out the string and its length in that case.
(Solution: *stringer.awk*)

- 3** Modify the answer from **2** to incorporate the functions into a user-defined function.
(Solution: *func.awk*)

- 4** Create a script which uses the **rand()** function to generate a random list of telephone numbers. You can do this with a shell wrapper holding two separate *awk* scripts, and use intermediate files to hold the results.
(Solution: *phonegen*)

