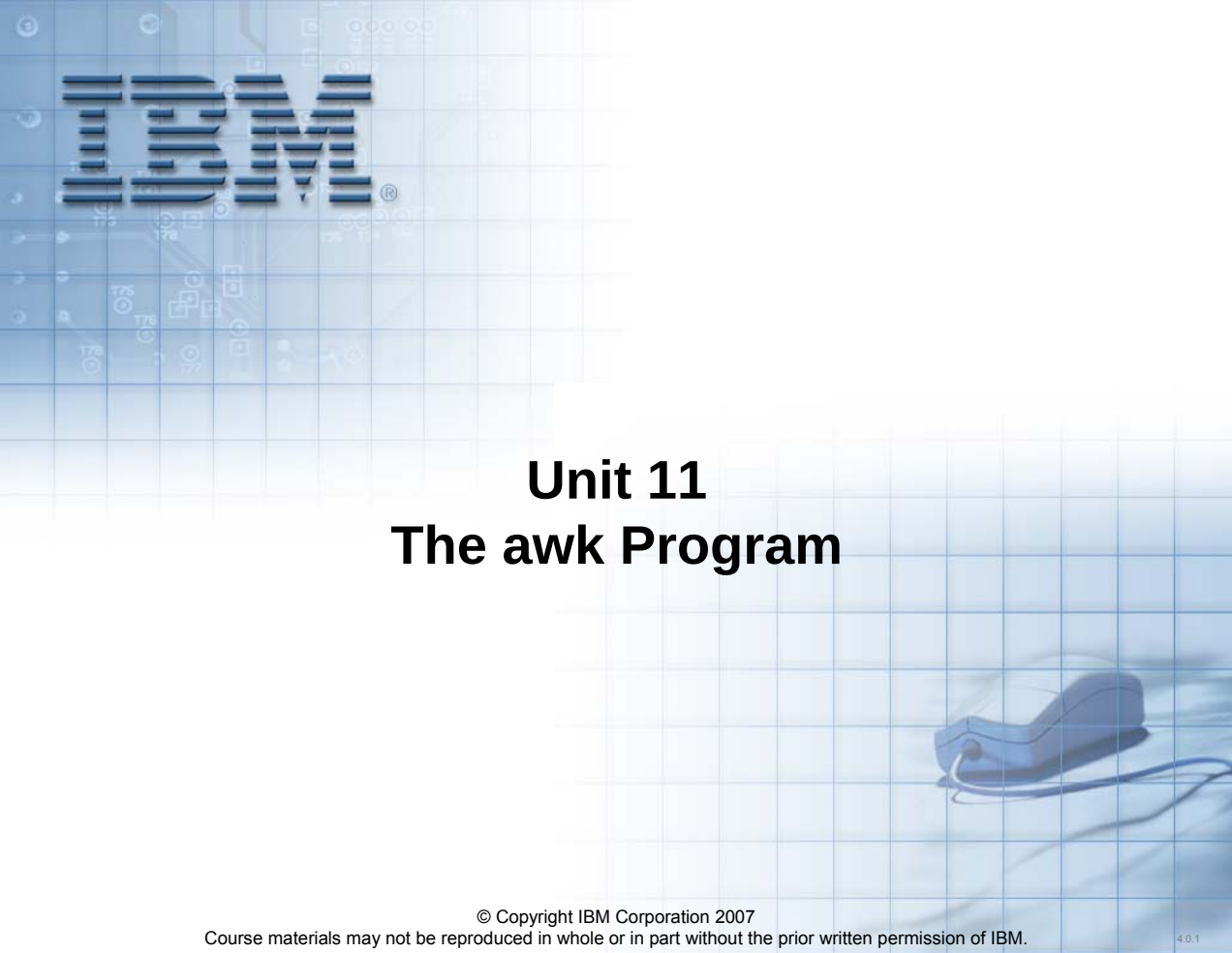


Unit 11
The awk Program

Unit 11 Objectives -- The awk Program

After completing this unit, you should be able to use the awk utility by looking at:

- Regular expressions in **awk**
- Basic **awk** programming
- **BEGIN** and **END** processing
- Flow control – **if**, **while** and **for**
- Leaving loops – **continue**, **next** and **exit**
- **awk** arrays
- Better printing
- **awk** functions

What Is Awk?

- **awk** is a programming language used to manipulate text
- **awk** sees data as words (**fields**) in a line (**record**)
- An **awk** command consists of a **pattern** and an **action** comprising one or more statements

```
awk  '/pattern/ { action }'  file  ...
```

- **awk** tests every **record** in the specified ***file(s)*** for a **pattern** match. If a match is found, the specified **action** is performed
- **awk** can act as a filter in a pipeline or take input from the keyboard (standard input) if no ***file(s)*** are specified

Sample Data – awk

Lastname,<Space>Firstname<Tab>nnn-mmmm

```
$ cat phone.list
Terrell, Terry              617-7989
Franklin, Francis           704-3876
Patterson, Pat              614-6122
Robinson, Robin             411-3745
Christopher, Chris          305-5981
Martin, Marty               814-5587
Llewellyn, Lynn             316-6221
Jansen, Jan                 903-3333
Llewellyn, Lee              817-8823
$ _
```

The same file is used in the RE and sed units

awk Regular Expressions

- Like sed, regular expressions are "**/**" delimited – "**/x/**"
- All of the previous regular expression metacharacters can be used with **awk**

awk has the following extensions

/x+/	for one or more occurrences of **x**
/x?/	zero or one occurrence of **x**
/x|y/	matches either "**x**" or "**y**"
(string)	groups a string – for use with **+** or **?**

Example:

```
/t[i|o]?n[iey]+/
```

matches: tiny, tony, toni, toney, tone, tny (and others...)

awk Command Syntax

- Basic syntax

```
pattern  { actions }
pattern
         {
          actions
         }
```

- Multiple statements in an action
  - Use a line break or a semi-colon

```
$ awk '/Ll/ { print $1 ; print $3 }' phone.list
```

- Comments start with a # until the end of a line

```
$ awk '/Ll/ { print $1 # prints field 1
> print $3 }' phone.list
```

The print Statement

One useful **action** is to **`print`** the data!

```
awk '/pattern/ { print }' ifile > ofile
```

`awk` tests each **`record`** of the input for the specified *pattern*

- When a match is found the **`print`** statement sends the entire **`record`** to standard output

awk Fields and Records

- Referencing fields in a record

  $0 = the entire record
  $1 = the first field in the record
  $2 = the second field in the record
  ...

- To print Jansen's phone number from phone.list:

```
$ awk '/Jansen/ { print $3 }' phone.list
903-3333
```

- To place that phone number into a variable:

```
$ JanNum=$(awk '/Jansen/ { print $3 }'  phone.list)
```

print Examples

- Special character sequences are available for use in print strings or regular expressions

`\n`	newline
`\t`	tab
`\r`	carriage return

```
$ awk '/^Ll/ { print "Name:\t", $1
>      print "Number:\t", $3, "\n" }' phone.list
Name:        Llewellyn,
Number:      316-6221

Name:        Llewellyn,
Number:      817-8823

$ _
```

Comparison Operators and Examples

To compare regular expressions or strings with values:

`==`	equal to	`!=`	not equal to
`<`	less than	`<=`	less than or equal to
`>`	greater than	`>=`	greater than or equal to
`~`	matched by RE	`!~`	not matched by RE
`||`	logical "or"	`&&`	logical "and"

Examples:

`$1 ~ /x/`	field one matches regular expression **x**
`$1 != "No"`	field one doesn't match string "**No**"

You can use comparison operators in the pattern to select records

```
$ awk '$1 == "Terrell," { print $2, "Smythe" }' phone.list
Terry Smythe
$ _
```

Arithmetic Operators

You can use the following operators to perform arithmetic:

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`%`	remainder
`^`	exponential (x^y, raise x to the power y)
`++x x++`	pre and post increment
`--x x--`	pre and post decrement
`=`	assignment (x = 4)

x **op=** y x = x **op** y

for: **+=, -=, *=, /=, %=**

Example

count **=** count **+** 2

count **+=** 2

User Variables and Expressions

You can define your own variables:

- Names must:
  - Start with a letter or underscore
  - Be followed by letters, underscores, or digits

- **awk** does not require variables to be defined before use

Variables are initialized as empty (numerically zero)
  - The empty string is null ("")
- Referenced by name only
- Can be passed through from the command line

BEGIN and END Processing

You have seen the **pattern** and **action** with **awk** syntax
You can also have actions at the beginning and end of input
You use the special patterns **BEGIN** and **END**

```
awk 'BEGIN { begin_action }
     pattern { action }
     pattern { action }
     END { end_action }'   file...
```

Where:

BEGIN means execute the *begin_action* before any input read

END means execute *end_action* once all input has been read

BEGIN without END Example

You can use `BEGIN` to print a header to the output...

Here we have a `BEGIN` with no `END`

```
$ awk  'BEGIN  { print "Words in phone.list"}
>              { wcount = wcount + NF
>               print wcount }' phone.list
Words in phone.list
    3
    6
    9
    ...
    24
    27
$ _
```

The statements within the second set of braces were performed on every line of "phone.list" as no pattern was specified

END without BEGIN Example

You can use **END** to print a trailer or summary after the output:

```
$ awk  '{ wcount = wcount + NF }
> END { print "Words in phone.list: ",
          wcount }' phone.list
Words in phone.list: 27
$ _
```

- The statement within the first set of braces refers to the main **action**
- The main **action** is performed on every line of the file "phone.list", so the final value of **wcount** holds the total number of fields (or words) in the file
- At the end of the input **END** actions are processed
- This prints the heading with the total word count

Built-In Variables

awk provides a number of useful built-in variables:

FILENAME	the name of the current ***file***
NF	total number of ***fields*** in the current record
NR	number of ***records*** encountered
FS	***input field separator*** (default is space or tab)
RS	***input record separator*** (default is newline)
OFS	***output field separator*** (default is space)
ORS	***output record separator*** (default is newline)

Built-In Variables Examples (1 of 2)

```
$ cat employee.list
Name, company, city, phone
Drew A. Chart, IBM, Wash. D.C., 202-555-3788
Wanda C. Results, IBM, Denver, 303-555-8068
Hyde N. Sikh, IBM, Atlanta, 404-555-3523
$ _


$ awk 'BEGIN { FS = "," ; OFS = ":" }
>         { print $1, $4 }' employee.list
Name: phone
Drew A. Chart: 202-555-3788
Wanda C. Results: 303-555-8068
Hyde N. Sikh: 404-555-3523
$ _
```

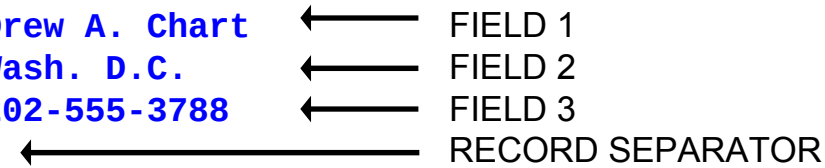
Built-In Variable Examples (2 of 2)

```
$ cat authors
Drew A. Chart
Wash. D.C.
202-555-3788

Wanda C. Results
Denver, CO
303-555-8068

Hyde N. Sikh
Atlanta, GA
404-555-3523

$ awk 'BEGIN { FS="\n" ; RS="\n\n" ; OFS="\n" ;
ORS="\n\n"}
> { print $1, $3
> } ' authors
```



if - else if - else Statement

Syntax:

```
awk '{
     if (first logical test) {
         action if test true
     }
     else if (second logical test)  {
         action if first test false and
         second test true
     }
     else  {
         action if both tests false
     }
}' file
```

Example:

```
$ awk '{
     { if ( $2 == "Terry" )
           print $2 ", " $1 "--" $3
     }
}' phone.list
```

The while Loop

Syntax:

```
awk   ' {
      while   (condition)        {
                 action
      }
      } ' file
```

Example:

```
awk   ' {i = 1
     while (i <= 4)
     {  print $i ; ++i }
     } ' file
```

The for Loop

Syntax:

```
awk  '{
       for (initialize; test; increment)
       {   action
       }
      }' file
```

Examples...

- To read and print each field of the current input line

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

- To print from the last field to the first of the current line

```
for (i=NF; i>=1; i--){
     print $i
   }
```

The break, continue and next Statements

The **continue** statement stops the current innermost loop iteration and starts the next one:

```
awk  '{
        y = 42
        for (x=1; x<=NF; x++) {
           if (y!=$x)
           {
              continue # break
           }
           print x, $x
           }
        }'  file
```

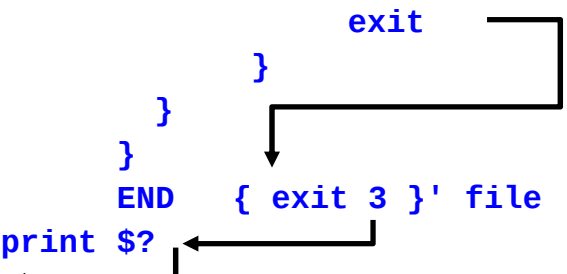
The **next** statement causes the next **record** to be read in, and the program to start from the first **pattern { action }** block again:

```
awk 'BEGIN { action }
   pattern {
            action
            action
            next
            action
           }
   END { action }' file
```

The exit Statement

The **exit** statement jumps to any **END** processing – or out of the program if already in the **END** section. An exit code can be passed back to the shell:

```
$ awk  '{
>         y = 42
>         for (x=1; x<=NF; ++x) {
>              if (y==$x) {
>                   print x, $x
>                   exit
>              }
>         }
>       }
>       END   { exit 3 }' file
$ print $?
3
$ _
```



Arrays

- **`awk`** allows **`array`** variables
- An **`array`** is a variable with an index
- An index is an expression in brackets
  - For example, **`array[10]`**
- **`awk`** arrays are **associative**
  - Index can be a string or number
  - No implicit order
  - To access all elements, use the **`in`** operator

    `for ( var in array_name )`
- Be aware that all **`array`** indices are internally strings

printf for Formatted Printing

- One use of **awk** is as a report generator
- Better printing formats required
  - Use **printf**
  - **printf** **syntax:** **printf (fmt [, args])**
- Parentheses are optional
- **fmt** is usually a string constant with format specifications
- Specifiers are like the C language **printf**
- Format specification: **%<char>**
  - **%s** string
  - **%d** decimal integer
  - **%f,%e** floating point (fixed or exponent notation)
  - **%o** unsigned octal
  - **%%** literal percent

printf Formats

- Format specification strings can use modifiers
  - **`%-width.precision`**
  - If width used, contents are right justified
  - Use **`-`** (minus/hyphen) after % to left justify
  - Precision controls
    - Number of digits to right of decimal point for numeric values
    - Maximum number of characters to print for string values
- To print Hello within #'s right justified in 10 character field
  - **`printf ("#%10s#\n", "Hello")`**
- To print a number left justified with minimum three characters
  - **`printf ("%-3d\n", $1)`**

Functions in Awk

- There are four types of functions
- Three types are built-in to **awk**
  - General
  - Arithmetic
  - String
- The fourth type is a user defined function
  - General functions include
    - Close
    - System
    - Getline

Built-In Arithmetic Functions

Functions available include:

`atan2(y,x)`	arctangent of y/x in range $-\pi$ to $+\pi$
`cos(x)`	cosine of x (x in radians)
`sin(x)`	sine of x
`exp(x)`	*e* to the power x
`log(x)`	natural log of x
`sqrt(x)`	square root of x
`int(x)`	truncated value of x
`rand()`	pseudo-random number r, $0 \leq r \geq 1$

Built-In String Functions

Functions available include:

length(s)	length of string **s** or of $0 if **s** not supplied
index(s,t)	position of substring **t** in **s** or zero if not present
match(s,r)	position in **s** of where RE **r** begins or zero
sub(r,s,t), **gsub(r,s,t)**	substitute **s** for **r** in **t**, returns 1 for OK uses $0 if **t** not supplied (gsub does all matches)
split(s,a,sep)	parses **s** into array **a** elements using field separator **sep** (use **RS** if not supplied)
Set by match() **RSTART**	start of the match (same as the return value)
RLENGTH	length of the matching sub-string

Built-In String Functions Examples

1 `awk ‘{print len($1)}’ myfile`

2 `awk ‘{print index($1, “a”)}’ myfile`

3 `awk ‘{print match($1, “i.a”)}’ myfile`

4 `awk ‘{match($1, “i.a”); print RSTART, RLENGTH]’ myfile`

5 `awk ‘{print gsub(/a/,”b”,$1), $0}’ myfile`

6 `awk ‘{gsub(/a/,”b”,$1)’ print $0}’ myfile`

7 `awk ‘{split ($0,var,”:”); print var[1], var[2], var[6]]’ /etc/passwd`

Checkpoint

1. With **awk**, what happens if I don't supply a **pattern**?

2. With **awk**, what happens if I don't supply the **action**?

3. **awk** causes the **-f** option to read instructions from a default line.

4. **awk** must have both the **BEGIN** and **END** statements. T or F

5. Using **awk**, have the output from the **df** command only show the **% used** and **mount point**.

Unit Summary

- Regular expressions in **awk**
- Basic **awk** programming
- **BEGIN** and **END** processing
- Flow control – **if**, **while** and **for**
- Leaving loops – **continue**, **next** and **exit**
- **awk** arrays
- Better printing
- **awk** functions