CST8177

awk

The **awk** program is not named after the sea-bird (that's auk), nor is it a cry from a parrot (awwwk!). It's the initials of the authors, Aho, Weinberger, and Kernighan.

It's known as a pattern-matching scripting language, and derives from **sed** and **grep**, who both have **ed**, the original Unix editor, as their ancestor.

We will use the GNU version of **awk**, known (of course) as **gawk** (there's even a version called **mawk**, for Microsoft platforms).

For convenience, both the **awk** and **gawk** names are supported by Linux, as links to the same program executable.

```
awk [ options ] -f program-file file ...
awk [ options ] program-text file ...
```

The **program-text** is always in the form:

```
[selection] { action }
```

and is most usually enclosed in single quotes.

The options need not be used. Some of the common ones include:

-F fs **--field-separator fs**

Use **fs** for the input field separator (the value of the **FS** predefined variable) instead of a space. See also the **$OFS** (output field separator) variable.

-v var=val **--assign var=val**

Assign the value **val** to the variable **var** before execution of the script begins.

-f program-file **--file program-file**

Read the source from the file **program-file**, instead of from the first command line argument. Multiple **-f** options may be used.

There are many more, but we will focus on these three.

Getting started

Let's try some **awk** on the password file. Since it uses '**:**' to separate fields, we'll have to use **-F ':'**.

```
[Prompt]$ awk -F ':' '{ print $1 }' /etc/passwd
root
bin
...
user1
```

Oops, rather too many. Now select only those with UIDs of 500 or more:

```
[Prompt]$ awk -F ':' '{ if ($3 >= 500) \
                          print $1}' /etc/passwd
nfsnobody
allisor
test1
test2
user2
```

Let's look at these two **awk** "programs":

```
awk -F ':' '{ print $1 }' /etc/passwd
```

There's our **-F** to change from the default separator (spaces or tabs) to the '**:**' we need, followed by **'{ print $1 }'** which is the program, and finally the filename we're working with, **/etc/passwd**.

The program is in single quotes, to keep the shell from interfering. Enclosed in curly brackets, we have a single statement, **print $1**. In **awk**, we refer to the tokens of an input line just like command-line arguments. The only difference is that **$0** refers to the whole line at once.

This program, therefore, tells **awk** to print just the first field, the user id (account name, whatever), from each line that matches the omitted regex (that is, all lines is the default selection).

The second **awk** program uses an **if** statement as well.

```
if ($3 >= 500) print $1
```

It looks reasonable enough: print the user id only if field 3 (the UID) is at least 500. That is, only print the user accounts (plus that peculiar **nfsnobody** that some of us have: it's UID on this system is **4294967294**).

We can also use a regex with **awk** to select the lines we want:

```
...]$ awk -F ':' '/^[^:]*:[^:]*:[5-9][0-9][0-9]/ \
                              { print $1 }' /etc/passwd
allisor
test1
test2
User2
```

That regex chooses all UIDs from 500 to 999. I know which of these I prefer.

Instead of a regex, you can use a relational expression:

```
[Prompt]$ awk -F ':' '$3 >= 500 && $3 < 1000 \
               { print $1 }' /etc/passwd
allisor
test1
test2
user2
```

As usual with Linux tools, **awk** has many ways to accomplish a result.

What would this look like as a script? As an **awk** file?

Here's an **awk** file execution. No execute permission is needed, since we call **awk** to process it.

```
[Prompt]$ awk -F ':' -f awk0 /etc/passwd
allisor
test1
test2
user2
```

Here is the **awk0** file:

```
$3 >= 500 && $3 < 1000 { print $1 }
```

And a corresponding bash script.

```
#! /bin/bash
cat /etc/passwd | while read line; do
    a3=$(echo $line | cut -d ':' -f 3)
    if (( $a3 >= 500 && $a3 < 1000 )); then
        echo $(echo $line | cut -d ':' -f 1)
    fi
    done
exit 0
```

Hmmm. Quite a difference, isn't there?

Oh, you want an executable file and for the file to be an argument? Then **chmod +x** this as **lu** (list users):

```
awk -F ':' \
    '$3 >= 500 && $3 < 1000 { print $1 }' $*
```

Now run **./lu /etc/passwd**

awk statements

An **awk** "program" is a series of statements, each of which can select lines with a regex pattern, a relational expression, or omit both to select all lines in the file. A regex or expression preceded by '**!**' is inverted, selecting those lines that do not match.

There are also special patters, like **BEGIN** and **END**, that match before the first read and after end-of-file. There are **&&** (AND) and **||** (OR) used to combine pattern elements or relational expressions.

The selection pattern (if any) is followed by a series of action statements inside a set of curly brackets. These are generally simpler that similar bash script statements.

Do you need to write PDL for an **awk** program? Yes, but only if it consists of more than a few patterns and/or actions. You may choose to write PDL in all cases so that you have a record of what you intended to do.

awk regex extensions

The regex expressions supported by **awk** are the extended form as supported by **egrep**, with some additional features supported particularly by **awk**:

- **\y** matches the empty string at the beginning or end of a word.
- **\B** matches the empty string within a word.
- **\<** matches the empty string at the start of a word.
- **\>** matches the empty string at the end of a word.
- **\w** matches any word-constituent character (letter, digit, or underscore).
- **\W** matches any character that is not part of a word..
- **\'** matches the empty string at the beginning or end of a buffer (string).

awk actions

Actions are enclosed in curly brackets **{}** and consist of the usual statements found in most languages. The operators, control statements, and input/output available are patterned after those in the C programming language.

You have already seen the use of **$0** and **$1**, **$2**, and so on, and you've seen a simple **if** statement. The full form is:

```
  if (conditional expression) statement-if-true \
      [else statement-if-false]
```

Combine several statements together in **{}** and use '**;**' to separate commands:

```
[Prompt]$ awk -F ':' -v i=0 \
 '/^test/ { if ($3 >= 500) { print $1; i++ } \
            else continue } \
 END      { print "i = " i  }' /etc/passwd
test1
test2
i = 2
```

`awk` operators and functions

The assignment operators are the same as bash: **`= += -= *= /=`**. You also have the normal arithmetic operators: **`+ - * / % ++ --`** (includes pre- and post- forms). The relational operators include the usual **`== != > < >= <=`** as well the new ones **`~`** and **`!~`** for regex matching/not matching (put the regex on the right side of a regex match only, within a pair of '`/`' characters). There are also **`()`** for grouping, the **`&& || !`** operators, " " (space) for string concatenation, plus others we won't likely use.

There are many pre-defined functions. A few of them are:

`gsub(r, s [, t])` For each substring matching the regular expression **`r`** in the string **`t`**, substitute the string **`s`**, and return the number of substitutions. If **`t`** is not supplied, use **`$0`**.

`sub(r, s [, t])` Just like **`gsub()`**, but only the first matching substring is replaced.

more functions

`index(s, t)` Returns the index of the string **`t`** in the string **`s`**, or 0 if **`t`** is not present. This means that character strings start counting <u>at one, not zero</u>.

`length([s])` Returns the length of the string **`s`**, or the length of **`$0`** if **`s`** is not supplied.

`strtonum(str)` Examines **`str`**, and returns its numeric value. If **`str`** begins with a leading **`0`**, or a leading **`0x`** or **`0X`**, it assumes that **`str`** is octal or hexadecimal.

`substr(s, i [, n])` Returns the substring of **`s`** starting at index **`i`**. If **`n`** is omitted, the rest of **`s`** is used.

`tolower(str)` Returns a copy of the string **`str`**, with all the upper-case characters in **`str`** translated to their corresponding lower-case counterparts. Non-alphabetic characters are left unchanged.

`toupper(str)` As for **`tolower()`**, but for upper-case.

awk control statements

```
if (condition) statement [else if statement] ...
                                [ else statement ]
while (condition) statement
do statement while (condition)
for (expr1; expr2; expr3) statement
for (var in array) statement
break
continue
delete array[index]
delete array
exit [ expression ]
{ statements }
statement ; statement
```

awk input statements

`getline` - get the next line from **`stdin`** into **`$0`**

`getline <` - get the next line from a re-directed file

`getline var` - get the next line into **`var`**

`cmd | getline [var]` - get lines from the **`cmd`**

`next` - Stop processing the current input record. The next input record is read and processing restarts from the first pattern

`nextfile` - Stop processing the current input file.

Like the bash **`while read`**, **`getline`** returns true (**`1`**) for good input, false (**`0`**) for end-of-file, or **`-1`** for an error.

Note that the true and false values are reversed from bash; the **`awk`** commands are adjusted as required so (for example) a **`while (getline new_line <$2)`** will still loop until end-of-file.

awk output statements

`print`	- print the current record to **`stdout`**
`print expr`	- print the expression(s) to **`stdout`**
`print >[>]`	- print/append to a re-directed file
`printf fmt`	- print the formatted record to **`stdout`**, or with **`>`** or **`>>`**, print or append to the re-directed file
`print |`	- print/append expression(s) to a pipe
`printf fmt |`	- print/append a formatted record to a pipe

Special file names

When doing I/O redirection from either **`print`** or **`printf`** into a file, or via **`getline`** from a file, **`awk`** recognizes certain special shell filenames internally. These filenames allow access to streams inherited from **`awk`**'s parent process (usually the shell).

These file names may also be used on the command line to name data files. These filenames are:

`/dev/stdin`	The standard input.
`/dev/stdout`	The standard output.
`/dev/stderr`	The standard error output.

Note that these may be used on the command line for any command, utility, built-in, script, or whatever; they are not specific to **`awk`**.

A useful awk script

Let us suppose that we've been given an assignment to write a script to list and sum file sizes for any given directory plus, at the user's discretion, its sub-directories.

```
START fsize
   PRINT column headers
   FOR each line from an ls command
      IF regular file
         ADD size to total
         COUNT file
         PRINT size and name
      ELSE IF directory
         PRINT "<dir>" and name
      ELSE IF line from -R
         PRINT *** and the line
      ENDIF
   END FOR
   PRINT total and file count
END fsize
```

```
ls -l $* | awk -v sum=0 -v num=0 '

BEGIN {   # before starting
    print "BYTES", "\t", "FILE" }

NF == 8 && /^-/ { # 8 fields and file
    sum += $5
    num++
    print $5, "\t", $8 }

NF == 8 && /^d/ { # 8 fields and dir
    print "<dir>", "\t", $8 }

NF == 1 && /^.*:$/ { # subdirectories
    print "***\t", $0 }

END {   # after end
    print "Total:", sum, "bytes in", num,
"files" }'
```

```
[Prompt]$ ./fsize.awk -R empty
BYTES    FILE
***      empty:
59       arf
36       awk0
58       awk0.1
198      awk1
<dir>    dir1
12       file1
12       file2
17       file3
10       not
***      empty/dir1:
23       file4
Total: 425 bytes in 9 files
```

An awk-ward shell script

```
#! /bin/bash
declare -a line
declare tot_bytes=0
declare tot_files=0
declare nf=0

# create a temporary file
declare temp=$(mktemp)

# put columns headers
echo -e "BYTES\tFILE"
```

```
ls -l $* | while read -a line; do
    nf=${#line[*]}
    if (( nf == 8 )); then
        if [[ "${line[0]:0:1}" == "-" ]]; then
            (( tot_bytes += ${line[4]} ))
            (( tot_files++ ))
            echo -e ${line[4]} "\t" ${line[7]}
        elif [[ "${line[0]:0:1}" == 'd' ]]; then
            echo -e '<dir>\t' ${line[7]}
        fi
    fi
    if (( nf == 1 )); then
        if echo ${line[0]} | grep -q '^.*:$'; then
            echo -e '***\t' ${line[0]}
        fi
    fi
# write intermediate values to temp file
    echo $tot_bytes $tot_files > $temp
done
```

```
# read back final intermediate values
read tot_bytes tot_files < $temp

# remove temporary file
rm -f $temp

# now print the totals
echo Total: $tot_bytes bytes in $tot_files files
```

```
[Prompt]$ ./fsize.sh -R empty
***      empty:
59       arf
36       awk0
58       awk0.1
198      awk1
<dir>    dir1
12       file1
12       file2
17       file3
10       not
***      empty/dir1:
23       file4
Total: 425 bytes in 9 files
```