Bash: AWK

Glenn Bruns
CSUMB

Lecture Objectives

After this lecture, you should be able to:

- Write simple awk programs

AWK

- AWK is a “little language” for text processing
- AWK = Aho/Weinberger/Kernighan
- Powerful and easy-to-use language
- Features include pattern matching and associative arrays
- Comes in several flavors, including ‘gawk’
- Essential tool for command-line power users

Example 1 – printing fields

```
$ ls -l > temp.txt

$ head -3 temp.txt
total 32
-rw-r--r--. 1 brun1992 shell_faculty  312 Oct  1 12:39 Makefile
-rw-r--r--. 1 brun1992 shell_faculty    0 Oct  1 13:50 temp.txt

$ awk '{ print $9","$5 }' temp.txt
,
Makefile,312
temp.txt,0
times.awk,131
times.csv,115
times.txt,419
tlb,6531
tlb.c,1848
tlb-start.c,772
```

$i refers to the ith field (starting at 1)

$0 refers to the entire input record

Example 2 – variables, END section

```
$ head -3 temp.txt
total 32
-rw-r--r--. 1 brun1992 shell_faculty  312 Oct  1 12:39 Makefile
-rw-r--r--. 1 brun1992 shell_faculty    0 Oct  1 13:50 temp.txt
$ awk '{sum += $5} END{print sum}' temp.txt
10128
$
```

you can also use a ‘BEGIN’ section

You don’t need to use $ in front of variables, like you do with bash.

We didn't need to initialize 'sum'.

What does this do?

```
{line = $0}

END {print line}
```

It prints the last line in the file.

Example 3 – awk built-in variables

```
$ head -3 temp.txt
total 32
-rw-r--r--. 1 brun1992 shell_faculty  312 Oct  1 12:39 Makefile
-rw-r--r--. 1 brun1992 shell_faculty    0 Oct  1 13:50 temp.txt
$ awk '{if (NF > 2) sum += $5} END{print sum}' temp.txt
10128
$
```

NF is a built-in variable – number of fields in input record

Some other built-in variables:

NR – number of current input record

FS – field separator character (default blank and tab)

What does this do?

```
{ print "a" }
{ print "b" }
```

Input file:

```
foo
bar
baz
```

output:

```
a
b
a
b
a
b
```

AWK processes a file line-by-line:

- for each line, starting with the first
  - run the AWK script on that line

What does this do?

```
{ print NR": "$0 }
```

It numbers the lines in the input file.

Example 4 – pattern matching

```
$ head -6 alloc-hw-1.txt
Free List [ Size 3 ]:  [ addr:1000 sz:4 ] [ addr:1004 sz:5 ] [
addr:1009 sz:91 ]

ptr[2] = Alloc(3)  returned 1000 (searched 3 elements)
Free List [ Size 3 ]:  [ addr:1003 sz:1 ] [ addr:1004 sz:5 ] [
addr:1009 sz:91 ]

Free(ptr[2]) returned 0
$ awk '/^ptr.*returned/ { print $5 }' alloc-hw-1.txt
1000
1009
1019
1000
```

pattern { action-statements }

One kind of pattern: /regular expression/

Example 5 – matching on a field

```
$ head -6 alloc-hw-1.txt
Free List [ Size 3 ]:  [ addr:1000 sz:4 ] [ addr:1004 sz:5 ] [
addr:1009 sz:91 ]

ptr[2] = Alloc(3)  returned 1000 (searched 3 elements)
Free List [ Size 3 ]:  [ addr:1003 sz:1 ] [ addr:1004 sz:5 ] [
addr:1009 sz:91 ]

Free(ptr[2]) returned 0
$ awk '$4 ~ /Size/ { print $5 }' alloc-hw-1.txt
3
3
4
4
4
3
4
```

Some other kinds of patterns:

- pattern `$4 == "Size"` succeeds if field 4 is exactly 'Size'
- pattern `$4 ~ /Size/` succeeds if field 4 contains 'Size'
- pattern `$4 !~ /Size/` succeeds if field 4 does not contain 'Size'

Structure of an AWK program

AWK programs mostly consist of pattern-action statements:

```
pattern { action-statements }
```

The pattern is optional.
Some typical patterns:

/^[0-9]/	line begins with a digit
! /---/	line does not contain three dashes
$1 == 'foo'	first field is 'foo'
$2 ~ /foo/	second field contains 'foo'
$4 !~ /baz/	fourth field does not contain 'baz'

Example 6 – awk scripts

```
$ cat addr.awk
# get address of returned memory
/^ptr.*returned/ {
   print $5
}
$ awk -f addr.awk alloc-hw-1.txt
1000
1009
1019
1000
1000
```

If you added this shebang line to the top of the script

```
#!/usr/bin/awk –f
```

and gave the script execute permission, you could write

```
$ ./addr.awk alloc-hw-1.txt
```

Example 7 – associative arrays

```
$ ps aux | head -3
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0   2900   708 ?        Ss   Sep12   2:37 /sbin/init
root         2  0.0  0.0      0     0 ?        S    Sep12   0:05 [kthreadd]
$ cat user-proc-cnt.awk
{
   cnt[$1]++
}
END {
   for (name in cnt) print name": "cnt[name]
}
$ ps aux | awk -f user-proc-cnt.awk | head -5
chan1722: 2
bb: 4
rpc: 1
brun1992: 7
bail5019: 2
$
```

When the awk program finished, we have:
cnt[chan1722] = 2,
cnt[bb] = 4,
cnt[rpc] = 1, etc.

Example 8 – detail of the script

```
{
  cnt[$1]++
}
END {
  for (name in cnt)
     print name": "cnt[name]
}
```

The index of an array can be any string.

Understand array cnt as a hash table.

string operators

index(s1, s2) – position of string s2 in s1; returns 0 if not present

length(s) - string length

split(s, a, c) - splits s into a[1],…,a[n] on character c, returns n

substr(s,m,n) – n-character substring of s starting at position m

how to find out more?

if statements

```
{
  x = $2
  if ($2 > 100) {
    print $2
  } else {
    printf("sum: %d\n", $2 + $3)
  }
}
```

variables don't have to be declared

'if' statements look like if statements in C

AWK statements don't need semicolons

AWK has printf

loops

```
{
  n = split($2, a, “/”)
  for (i = 1; i <= n; i++) {
    print a[i]
  }
}

$ cat temp.txt
bruns  /home/CLASSES/brunsglenn
$ awk -f looptest.awk temp.txt

home
CLASSES
brunsglenn
$
```

‘for’ statements look like if statements in C

using command-line arguments

```
{
  i = FIELD
  print $i
}
```

FIELD happens to be passed in from command-line

Note: we can use a variable when using $i

```
$ cat temp1.txt
foo 55  2
bar 160 3
$ awk -f testfield.awk -v FIELD=1 temp1.txt
foo
bar
$ awk -f testfield.awk -v FIELD=2 temp1.txt
55
160
```

Summary

- AWK is a little language for text processing
- key features include pattern matching and associative arrays
- AWK is fast; a typical use is to preprocess files with > 1 million lines
- Commands introduced in this lecture:
  - awk