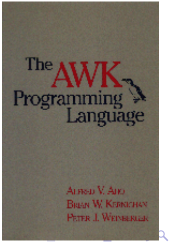
AWK – Introduction

Kurt Schmidt

Dept. of Computer Science, Drexel University

October 19, 2017



Intro

AWK

- Works well with record-type data
- Reads input file(s) a line (record) at a time
- Parses each record into fields
- Performs actions on each record that matches a given test
- Rich math and string libraries

Common Uses

- Data validation
  - Does every record have the same number of fields?
  - Do values make sense (negative values, correct type, etc.)
- Calculations
- Filtering out/reformatting certain fields
- Searches
  - Who got a zero on lab 3?
  - Who got the highest grade?
- Many others

AWK versions

Note, on your system `awk` might be linked to one of these:

`awk`	Original implementation of AWK
`nawk`	New AWK. Sorta AWK 2.0
`gawk`	Gnu’s implementation of AWK
`mawk`	A very fast implementation of AWK

There are others.

Invocation

- Can write short programs right on the commmand line (*very* handy)
  - print the 3rd field of every line:

```
$ awk ’{print $3}’ input.txt
```

- To execute an awk script:

```
$ awk -f script.awk input.txt
```

- Or, use this sha-bang, make the script executable:[1]

```
#!/usr/bin/awk -f
```

[1] Not so portable. Tough to get `env` to work here.

AWK Programs

Processing Input

- Input file(s) are read, a record (line) at a time
  - Files named on the command line are processed, in order
  - If no arguments are given, `stdin` is read
- Each line is checked against each pattern (test), in order of appearance
- For each pattern that matches, corresponding actions are performed on that record

Form of an AWK Program

An AWK program is a sequence of function definitions and one or more *rules* :

pattern {actions}

- *pattern* – Text. Numeric or string relational operator, or regular expression match
    - If empty, actions are applied to *every* record
- *action* – Statement or sequence of statements
    - If empty, default action is to print the entire line

Patterns

- Any numeric or string relational operator
- Normal logical operators, too (`&&` `||` `!`)
- AWK also has regular expression matches
  - REs are delimited by `/ /`

```
$3>0   # print all lines where field 3 is greater than 0
$1=="Ben"  # Find Ben's record
/[Zz]+czc/  # print all lines that contain a match for RE
$5~/[Ww]aldo/  # print record if Waldo is hiding in field 5
```

Patterns

- There are two special patterns:
  - BEGIN – True before first record is parsed
  - END – True after last record is parsed
- Empty test applies to every record

```
BEGIN { print "This happens once, before records are read" }
{ sum += $1 } # Sum all values in column 1
END {
  printf( "Read %d records. Sum of column 1 is %f\n", NR, sum )
}
```

```
BEGIN { FS="," } # Change field separator, parse CSVs
$4==100 { cnt += 1 }
END {
  printf( "%d students got 100% on midterm.\n", cnt )
}
```

Actions

- One or more AWK statements
- Default action is to print entire record ($0)

```
NR==1 { # Assume first record is column headers
   for( i=1; i<=NF; ++i )
      print i, $i   # show headers
}
  # print name, studID for section 2
NR>1 && $2=="002" { print $4,$5,$7 }
```

```
BEGIN { FS="," } # Change field separator, parse CSVs
$4==100 { cnt += 1 }
END {
   printf( "%d students got 100% on midterm.\n", cnt )
}
```

AWK Language

AWK is a C-like Scripting Language

- Syntax is rather C-like
  - Same keywords, branches, loops, operators
- Only 2 types: numbers (floats) and strings
- Variables are dynamically typed – no declarations
- Line comments begin with #
- Statements are separated by newline, or semicolon (`;`)
- Arrays are associative

Fields

- Fields are split over `FS`
  - By default, split over arbitrary whitespace
- Fields are identified by `$1 $2 $3 ... $NF`
- `NF` holds the number of fields in the current record
- `$0` is the current record

```
{print $0} # print entire line
{print $1, $3} # print 1st & 3rd field of each record
```

Variables

- Not declared – dynamic typing
- Same rules for naming identifiers as C, Java, etc.
- $\$n$ refers to the n^{th} field, where n evaluates to some integer

```
{
  for( i=1; i<=NF; ++i )
    print i, $i  # enumerate, print each field in the record
}
```

- Variables are either all global, or local to the function they're defined in

Some Built-in Variables

`NF`	Number of fields in current record
`NR`	The number of records read (so far)[1]
`FNR`	The number of records read in this file (so, the line #)
`FS`	Input field separator
`OFS`	Output field separator
`RS`	Input record separator
`ORS`	Output record separator

[1] Continual count, across files

Numbers

- Numbers, and arithmetic operators, are all float type
- Modulus (%) is an fmod operation
  - $b * int(a/b) + (a\%b) == a$ always holds
- There is an `int()` cast
- AWK uses ^ and `**`[1] for exponentiation
- AWK uses functions for bit-wise operations:

  `and compl lshift or rshift xor`
- Same increment, decrement, and op-assn operators

[1] Don't use the latter one

String Concatenation

- Accomplished simply by juxtaposition
- Might be helpful to put parentheses around numbers to be concatenated on to a string

```
$ awk 'BEGIN {print -12 " " -24}'
-12-24
$ awk 'BEGIN {print -12 " " (-24) }'
-12 -24
```

- Also a good idea to put parentheses around concatenated expressions

```
a = "some"
b = "file"
print "And I'm spinning..." > a b
```

vs.

```
print "And I'm spinning..." > (a b)
```

String Library Functions

`length` `tolower` `toupper`	Typical string functions
`index` `match`	Finding substrings[1]
`substr`	Pulling out substrings
`sub` `gsub` `gensub`	Search and replace
`split` `patsplit`	Return an array of strings
`sprintf`	Returns a formatted string
`strtonum`	Pulls numeric value from string

[1] Strings can not be indexed directly

Arrays

- All arrays are *associative*
  - Keys can be numbers or strings
  - Vectors can be sparse

a1.awk

```
BEGIN {
  a[1]="kurt"
  a[2]=13
  a[3]="Ski"
  for( k in a )
    print k, a[k]
}
```

output:

```
1 kurt
2 13
3 Ski
```

invocation:

```
$ awk -f a1.awk
```

Arrays - Sparse

- Vectors can be sparse
- Indices can even be negative
- Unitialised indices evaluate to `0` or `""`, depending on context

sparse.awk

```
BEGIN {
  a[5] = "kurt"
  a[12] = 13
  a[13] = "Ski"
  a[-77] = "I'm here, too"
  for( k in a )
    print k, a[k]
  print "a[7] =", "" a[7]
  print "a[7]+5 =", a[7]+5
}
```

output:

```
5 kurt
12 13
13 Ski
-77 I'm here, too
a[7] =
a[7]+5 = 5
```

Arrays - Associative

- Remember, they're all associative arrays
- Indices can be numbers or strings
  - In fact, they're *all* strings

tally.awk

```
{ tally[ $1 ] += $2 }

END {
  for( n in tally )
    print n, tally[n]
}
```

tally.simple:

```
Hannah,6
Sean,38
Marek,40
Hannah,40
Marek,36
Marek,37
Sean,26
Hannah,9
Morgan,27
Marek,49
```

output:

```
$ awk -F',' -f tally.awk < tally.sample
Morgan 27
Marek 162
Sean 64
Hannah 55
```

Arrays - More

- `delete` removes an array entry

```
delete a[3]
```

- AWK supports multidimensional arrays

```
a[i,j] = i*j
```

  - On a simple array
  - Subscripts are concatenated

- Gawk supports arrays of arrays:

```
a[i][j] = i*j
```

Sorting Arrays

Gawk provides sorting functions for arrays:

```
asort( src [, dest] )
asorti( src [, dest] )
```

- `asort` – Sorts the *values*
- `asorti` – Sorts the *indices* (keys)
- Returns the number of elements
- Destroys indices
  - Enumerates from 1

Arrays – asort

sort.awk

```
BEGIN {
  IGNORECASE = 0 + ic
  s = "shamrock bog clock craic Sidhe taisca eejit gob"
  split( s, a )
  n = asort( a )
  for( i=1; i<=n; ++i )
    print i, a[i]
}
```

output:

```
$ awk -f sort.awk
1 Sidhe
2 bog
3 clock
4 craic
5 eejit
6 gob
7 shamrock
8 taisca
```

Sorting – IGNORECASE

sort.awk

```
BEGIN {
  IGNORECASE = 0 + ic
  s = "shamrock bog clock craic Sidhe taisca eejit gob"
  split( s, a )
  n = asort( a )
  for( i=1; i<=n; ++i )
    print i, a[i]
}
```

output:

```
$ awk -v ic=1 -f sort.awk
1 bog
2 clock
3 craic
4 eejit
5 gob
6 shamrock
7 Sidhe
8 taisca
```

Numeric Sorting

There are a number of predefined predicates for `PROCINFO["sorted_in"]`, or you can define your own.

iSort.awk

```
BEGIN { PROCINFO["sorted_in"] = val_num_asc }
{ a[NR] = 0 + $1 } # make it a number
END { asort( a ) ; for( i in a ) print a[i] }
```

iSort.in:

```
66
28
13
76
31
4
75
59
```

output:

```
$ awk -f iSort.awk iSort.in
4
13
28
31
59
66
75
76
```

Sorting Non-destructively

- Remember, the original indices are lost
- `asort` takes an optional destination array
- Sorts into this array, leaving original untouched

iSort.awk

```
{ a[$1] = $2 }
END {
  n = asort( a, b )
  print "Original array:"
  for( k in a )
    print k, a[k]
  print "\nSorted values:"
  for( i=1; i<=n; ++i )
    print b[i]
}
```

Sorted Keys – `asorti`

Use `asorti` to sort the keys (indices) into another array[1]:

sorti.awk

```
{ a[$1] = $2 }

END {
  n = asorti( a, keys )
  for( i=1; i<=n; ++i )
    printf( "%10s : %s\n", keys[i], a[keys[i]] )
}
```

2-cols.in:

```
Sam,Into the Mystic
Roy,Gambol
Uri,Wine Down
Elisabeth,Bay Poet
Bruce,Legacy
```

output:

```
$ awk -F’,’ -fsorti.awk < 2-cols.in
     Bruce : Legacy
 Elisabeth : Bay Poet
       Roy : Gambol
       Sam : Into the Mystic
       Uri : Wine Down
```

[1] Not all AWKs, I think. You can write your own.

Defining Functions

- Introduced with keyword `function`

```
function func_name([param_list])
{
    body
}
```

- Parameters may not have same name as built-in variables
- Parameters may not have same name as function
- Parameter list contains arguments *and* local variables
  - Parameters not assigned are local variables, defaulting to the empty string
  - Variables in body not in parameter list are global

Functions – Example

func.awk

```
function foo( x, i )
{
  i = 12
  print "You passed in:", x
  j = "foo's j"
  k = "foo's k"
}

BEGIN {
  i = 5 # Also global
  j = "global"
  foo( "Heather" )
  print "i is:", i
  print "j is:", j
  print "k is:", k
}
```

output:

```
$ awk -f func.awk
You passed in: Heather
i is: 5
j is: foo's j
k is: foo's k
```

Functions – Example w/`return`

catmult.awk

```
function catmult( s, n )
{
   rv = s
   for( i=1; i<n; ++i )
      rv = rv(s)

   return rv
}

BEGIN {
   s = catmult( "Heather", 3 )
   print "s is:", s
}
```

output:

```
$ awk -f catmult.awk
s is: HeatherHeatherHeather
```

One-Liners

AWK One-Liners

- Line count

```
$ awk ’END {print NR}’
```

- Like `grep`

```
$ awk ’/regex/’
```

- Like `head`

```
$ awk ’FNR<=10’
```

- Add line numbers

```
$ awk ’{print FNR, $0}’
$ awk ’{printf( "%03d %s\n", FNR, $0 )}’
```

- Print lines 12-23, inclusive

```
$ awk ’FNR==12,FNR=23’
```

More One-Liners

- Remove blank lines[1]

```
$ awk '/./'
$ awk 'NF>0'
```

- Double-space a file

```
$ awk '1;{print ""}' # 2 separate rules
$ awk 'BEGIN{ORS="\n\n"};1' # Also 2 rules
```

- Smarter double-space

```
$ awk 'NF>0 {print $0 "\n"}'
```

- Remove leading whitespace from each line

```
$ awk '{sub(/^[ \t]+$/, "")};1' # Again, 2 rules
```

[1] Mind parsing DOS files. \n is preceded by \r