12 – Awk / Gawk

CS 2043: Unix Tools and Scripting, Spring 2019 [1]

Matthew Milano
February 18, 2019
Cornell University
Table of Contents

1. AWK / GAWK

2. More about the filesystem
As always: Everybody! ssh to wash.cs.cornell.edu

• Quiz time! Everybody! run **quiz-02-18-19**
• You can just explain a concept from last class, doesn’t have to be a command this time.
• NOTE: demos for this lecture:
  /course/cs2043/demos/12-demos
    • the leading / is important!
AWK / GAWK
**awk** Introduction

• **awk** is a programming language designed for processing text-based data.
  • Allows easy operation on fields rather than full lines.
  • Works in a *pattern-action* manner, like **sed**.
  • Supports numerical types (and operations).
  • Supports control-flow (e.g., **if** - **else** statements).

• Created at Bell Labs in the 1970s.
  • Alfred Aho, Peter Weinberger, and Brian Kernighan
  • An ancestor of **perl**, a *cousin* of **sed**.
  • Kernighan and Ritchie also invent C

• **Very** powerful.
  • It’s *Turing Complete*!
  • ... a lot of things are.
• **gawk** is the GNU implementation of the **awk** programming language.
• On BSD/OSX, it is just called **awk**.
• On GNU, it is technically **gawk**, but should reliably be symlinked as **awk**.
• There are many different implementations of the AWK programming language.
  • If you use C or C++, this is similar to how there are different compilers. The compiler is an “implementation” of the language (big quotes on that...).
  • If you use Python, it’s like the difference between CPython, PyPy, Jython, etc.
  • Different implementations of the same programming language.
• **awk** allows filters to handle text easily.

• The basic structure of an **awk** program is:

```c
pattern1 { commands1 }
pattern2 { commands2 }
# ...
```

• Patterns can be regular expressions!
  
  • Proceeds line by line, checking each pattern one by one.
  
  • If the pattern is found, the `{ commands }` are executed.
  
  • So for the above:
    
    • First line of input grabbed.
    
    • `pattern1` checked, if match `{ commands1 }` executed.
    
    • `pattern2` checked, if match `{ commands2 }` executed.
    
    • Next line of input grabbed.
    
    • Check `pattern1`, then `pattern2`, so on and so forth...
Why use **awk** over **sed**?

- Processing numerical values in **awk** is much more convenient.
- Variables and control flow in the actions.
- Convenient way of accessing fields within a given line.
- Flexible printing.
- Built-in arithmetic and string functions.
- Traditionally, **awk** has been used a lot in the scientific community e.g., biologists would use **awk** as a way of processing data and creating new table entries or something.
  - Basically, **awk** used to be the only real good *and* convenient option to process a large amount of data while still needing to perform mathematical computations or transformations.
  - These days there are many other options, but if you join a lab you may very well find some **awk** scripts creeping around and need to maintain them.
Simple Examples

• Print all lines containing Monster or monster.
  ```bash
  awk '/[Mm]onster/ {print}' frankenstein.txt
  ```

• If no action specified, default is to print the whole line.
  ```bash
  awk '/[Mm]onster/' frankenstein.txt
  ```

• The $0 variable in awk refers to the whole line.
  ```bash
  awk '/[Mm]onster/ {print $0}' frankenstein.txt
  ```

• First field (delimited by whitespace, or change field separator).
  ```bash
  awk '/[Mm]onster/ {print $1}' frankenstein.txt
  ```

• awk understands extended regular expressions by default :) 
  • We don’t need to escape +, ?, etc!
awk Shebang and BEGIN / END

- **awk** allows us blocks of code to be executed only once, at the beginning / end.
- With demo file `monstrosity.awk` and data file `frankenstein.txt` in current directory:

```bash
#!/usr/bin/awk -f
BEGIN { print "Starting search for monster..." }
/[Mm]onster/{ count++ } # Increment if [Mm]onster found
END { print "Found " count " monsters in the book." }
```

- Use the `-f` in the shebang to tell **awk** it expects a script.

```bash
$ ./monstrosity.awk   # hangs... no input file
$ ./monstrosity.awk frankenstein.txt # yay!
# shebang '#!/usr/bin/awk -f' makes same as ...
$ awk -f monstrosity.awk frankenstein.txt
```
Using Variables in **awk**

- words are *variables by default*
  - opposite of **bash**, where words are strings by default
  - **word** is a variable (**$word** works too)

- actions separated by semicolon
  - `{x = 0; y = 3; z = x + y; print z}`

- Not particularly whitespace sensitive!
Important Variables

- **NF**: the number of fields in the current line.
- **NR**: the number of lines read so far.
  - You cannot change NF or NR
- **FILENAME**: the name of the input file.
- **FS**: the *field separator*.
  - Example: change `FS=",""` for processing a comma-separated-value sheet.
  - Can also specify `-F` flag (capital!) to set the FS.
Pattern Matching with awk

- **awk** can match any of the following pattern types:
  - `/regular expression/`
  - `relational expression`
  - `pattern1 && pattern2`
  - `pattern1 || pattern2`
  - `pattern1 ? pattern2: pattern3`
    - If `pattern1`, then match `pattern2`. Otherwise, match `pattern3`
  - `(pattern)`: parenthesis to group / change order of operations.
  - `! pattern` to invert `pattern`
  - `pattern1, pattern2`: match `pattern1`, work on every line until matches `pattern2`
    - So you cannot combine this...
there are *many* match-action programming languages.

- `sed`
- `iptables`
- `firewalls`
- `datalog/prolog`

usually has *precedence*

- take first match, like `case`

`awk` does *not* have precedence
• Regular expression usage / comparison available here.
• Many more comparison operations detailed here.
• A wealth of useful / powerful built-in functions:
  • `toupper(x)`: make string upper case
  • `tolower(x)`: make string lower case
  • `exp(x)`: exponential of x
  • `rand()`: random number between 0 and 1
  • `length(x)`: the length of x
  • `log(x)`: returns the log of x
  • `sin(x)`: returns the sine of x
  • `cos(x)`: returns the cosine of x
  • `int(x)`: convert
  • etc.
• Much more information available here.
More about the filesystem
Inode the ultimate

• a data structure in a Unix-style file system that describes a file-system object such as a file or a directory.
• stores the attributes and disk block location(s) of the object’s data.
• attributes may include metadata
  • (times of last change, access, modification)
  • owner and permission data.
• Directories are lists of names assigned to inodes.
• A directory contains an entry for itself, its parent, and each of its children.
• This was all cribbed straight from wikipedia. Go look!
Links in the filesystem

- When and **inode** for an object is in a directory, we say it’s been **linked** into the filesystem tree
- the **ln** command makes and manages links.

```bash
ln [flags] <source> <target>
```

- works like **cp**; from **src** to **dst**
- creates a **peer** link; no notion of “original”
- only works on files
Symlinks in the filesystem

- A “soft” link or “symbolic” link isn’t a link at all
- works like a “shortcut” (really a junction) on Windows
- just a special file that contains a path in it
- looks light-blue under `ls`

Symlink a filesystem object into the directory tree

```
ln -s [flags] <source> <target>
```

- technically the same command as `ln`, but used very differently with the `-s` flag!
- creates a subordinate link; refers to the path.
- doesn’t check to see if the source path was sensible first!
- works on files or directories.
[1] Stephen McDowell, Bruno Abrahao, Hussam Abu-Libdeh, Nicolas Savva, David Slater, and others over the years. “Previous Cornell CS 2043 Course Slides”.