CS265
Advanced Programming Techniques

AWK

AWK, 1977

- Alfred **A**ho, Peter **W**einberger, Brian **K**ernighan
- AWK is a programming language
- Very good for
  - Simple, mechanical data manipulations
  - Calculations
  - Change format
  - Check data validity
  - Find items with some properties
  - Add numbers
  - Print reports
- Uses extended regular expressions
- Very short programs (often one liners!)
- AWK – is also a utility that takes as input an AWK program

Open Group Specification

https://pubs.opengroup.org/onlinepubs/009695399/utilities/awk.html

Java

```
import java.io.*;
import java.util.*;

public class demo {

    public static void main(String [] args) {
        String line;
        try {
            Scanner fs = new Scanner( new FileReader( "people.txt" )) ;

            while (fs.hasNextLine() ) {
                line = fs.nextLine();

                Scanner ls = new Scanner(line).useDelimiter(",");
                ArrayList<String> fields = new ArrayList<String>();
                while (ls.hasNext()) {
                    fields.add(ls.next());
                }
                if ((fields.get(2)).contains("@gmail.com")) {
                    System.out.println(line+" matched");
                }
            }
        }
        catch (FileNotFoundException e) {
            System.out.println("File not found.");
        }
    }
}
```

Python

```
import re

f=open("people.txt", "r")

if f.mode == 'r':
    lines = f.readlines()

for i in lines:
    i = i.rstrip()
    m = re.search("gmail\.com", i)
    if m:
        print(f"{i} matched")
```

awk

```
awk -F, '/@gmail.com/ {printf("%s matched\n", $0)}' people.txt
```

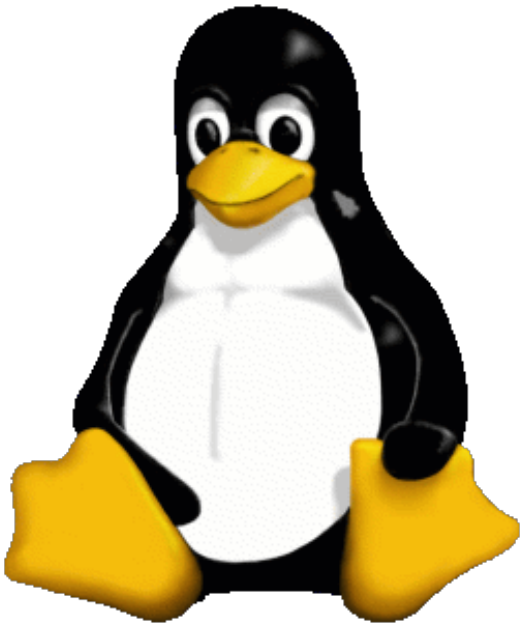
awk versions

Note, on your system awk may be linked to one of these

- awk Original implementation of awk from AT&T
- nawk New awk, like awk 2.0, also from AT&T
- gawk Gnu's implementation of awk (what Linux uses)
- mawk A very fast implementation of awk
- jawk A java version

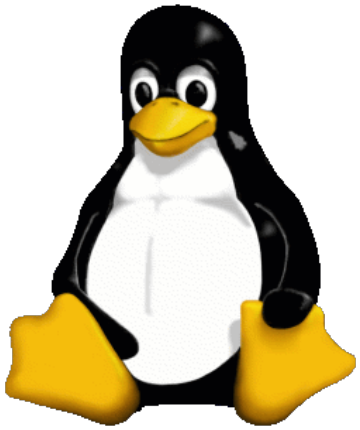
- There are others

What's my name?



Tux – The Linux penguin

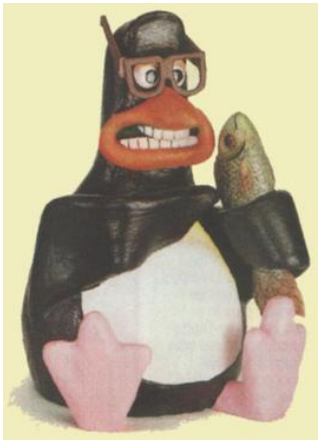
- Tux is a penguin character
- The official brand character of the Linux kernel
- Originally created as an entry to a Linux logo competition



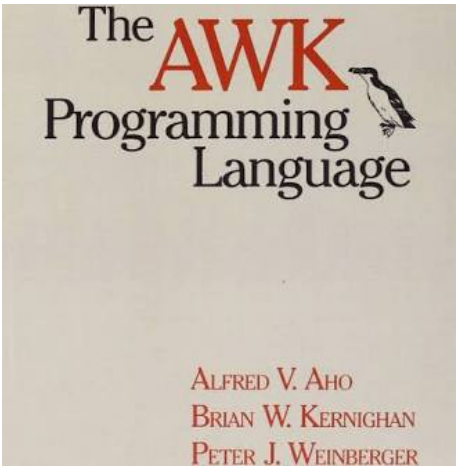
Tux

(T)orvalds (U)ni(X)"

1996



Linus Torvalds Favorite Picture



1988 AWK book

1988 AWK book

AWK

- AWK reads input file one record at a time
- AWK searches an input file for lines that match a pattern
  - Regular expressions
  - Comparisons on strings, numbers, arrays, variables, etc.
- For every matching line, a corresponding action is performed
- AWK splits the input line into fields automatically

```
pattern { action }
```

Getting started

- Supposed we have the `emp.data` file (name, payrate, hours)

```
Beth    4.00    0
Dan     3.75    0
Kathy   4.00    10
Mark    5.00    20
Mary    5.50    22
Susie   4.25    18
```

$1 name
$2 payrate
$3 hours

- What does this do?

```
awk '$3 > 0 { print $1, $2 * $3 }' emp.data
```

Getting started

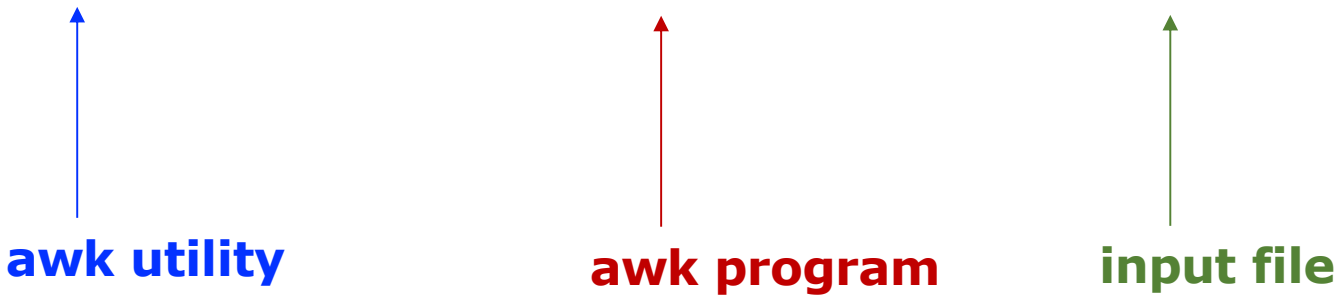
- Supposed we have the `emp.data` file (name, payrate, hours)

```
Beth    4.00    0
Dan     3.75    0
Kathy   4.00    10
Mark    5.00    20
Mary    5.50    22
Susie   4.25    18
```

$1 name
$2 payrate
$3 hours

- What does this do?

```
awk '$3 > 0 { print $1, $2 * $3 }' emp.data
```



Getting started

- Supposed we have the `emp.data` file (name, payrate, hours)

```
Beth    4.00    0
Dan     3.75    0
Kathy   4.00    10
Mark    5.00    20
Mary    5.50    22
Susie   4.25    18
```

$1 name
$2 payrate
$3 hours

```
awk '$3 > 0 { print $1, $2 * $3 }' emp.data
```

pattern to match
(for each line)

action to do
(for each line that matches)

- `$1` is the first field of each line, `$2` the second, `$3` the third
- It outputs the earnings for employees who worked

Getting started

- Supposed we have the `emp.data` file (name, payrate, hours)

```
Beth    4.00    0
Dan     3.75    0
Kathy   4.00    10
Mark    5.00    20
Mary    5.50    22
Susie   4.25    18
```

$1 name
$2 payrate
$3 hours

```
awk '$3 > 0 { print $1, $2 * $3 }' emp.data
```

pattern to match

action to do

- `$1` is the first field of each line, `$2` the second, `$3` the third
- It outputs the earnings for employees who worked

```
Kathy   40
Mark    100
Mary    121
Susie   76.5
```

Getting started

- Supposed we have the `emp.data` file (name, payrate, hours)

```
Beth    4.00    0
Dan     3.75    0
Kathy   4.00    10
Mark    5.00    20
Mary    5.50    22
Susie   4.25    18
```

- What does this do?

```
awk '$3 == 0 { print $1 }' emp.data
```

Getting started

- Supposed we have the `emp.data` file (name, payrate, hours)

```
Beth    4.00    0
Dan     3.75    0
Kathy   4.00    10
Mark    5.00    20
Mary    5.50    22
Susie   4.25    18
```

- What do this do?

```
awk '$3 == 0 { print $1 }' emp.data
```

- Lists names of people who did not work

```
Beth
Dan
```

awk programs

- Each awk program is a sequence of pattern/actions

```
pattern1 { action1 }
pattern2 { action2 }
…
```

- The pattern or action may be missing

```
$3 == 0 { print $1 }
$3 == 0
{ print $1 }
```

awk programs – BEGIN and END

- Two important patterns are specified by the keyword BEGIN and END
- They define actions that are done before or after a program executes

```
BEGIN { begin-action }
pattern1 { action1 }
pattern2 { action2 }
…
END { end-action }
```

- e.g., example

```
BEGIN {print "First Column" }
{print $1}
END {print "Done!" }
```

```
BEGIN {x=5}
{print $1 + $x}
END {print "Done!" }
```

How to run AWK

- From the command line

```
awk program inputFiles
```

```
awk '$3 == 0 { print $1 }' file1 file2
```

- From the command line without an input file

```
awk 'program'
```

- Using a file as input

```
awk -f programfile optional list of input files
```

- From a bash file

```
!/bin/bash
```

```
awk '$3 == 0 { print $1 }' file
```

Note the single quotes (why?)

Command-Line Syntax

```
awk [options] 'script' var=value file(s)
awk [options] -f scriptfile var=value file(s)
```

Standard Options

`-F fs` set the field separator to fs

`-v var` *value* Assign a *value* to variable *var*. This allows assignment before the script begins execution.

Example

```
awk -F: '{ print $1; print $2; print $3 }' /etc/passwd
```

AWK built-in variables

- AWK automatically splits the input lines into fields

```
$1 $2 $3 ..
```

- AWK special variables

`$0`	refers to the entire line
`FS`	input field separator (default space)
`OFS`	output field separator
`NF`	number of fields in current record
`NR`	number of records
`FNR`	number of current record (so, the line #)
`RS`	record separator (default newline)
`ORS`	output record separator
`FILENAME`	the current filename being read (there may be many)

There are more.

What do these programs do?

```
{ print }
```

What do these programs do?

`{ print }` print every line

```
{ print  $0 }
```

What do these programs do?

`{ print }` print every line

`{ print $0 }` also print every line

```
{ print $1, $2 }
```

What do these programs do?

`{ print }` print every line

`{ print $0 }` also print every line

`{ print $1, $2 }` print two fields with a field separator between the two

`{ print $1 $2 }`

What do these programs do?

`{ print }` print every line

`{ print $0 }` also print every line

`{ print $1, $2 }` print two fields

`{ print $1 $2 }` print one field by concatenating two fields

`{ print NF, $1, $NF }`

What do these programs do?

`{ print }` print every line

`{ print $0 }` also print every line

`{ print $1, $2 }` print two fields

`{ print $1 $2 }` print one field by concatenating two fields

`{ print NF, $1, $NF }` NF=number of fields, $NF = last field

`{ print $1, $2 * $3 }`

What do these programs do?

`{ print }` print every line

`{ print $0 }` also print every line

`{ print $1, $2 }` print two fields

`{ print $1 $2 }` print one field by concatenating two fields

`{ print NF, $1, $NF }` NF=number of fields, $NF = last field

`{ print $1, $2 * $3 }` compute and print

`{ print NR, $0 }`

What do these programs do?

`{ print }` print every line

`{ print $0 }` also print every line

`{ print $1, $2 }` print two fields

`{ print $1 $2 }` print one field by concatenating two fields

`{ print NF, $1, $NF }` NF=number of fields, $NF = last field

`{ print $1, $2 * $3 }` compute and print

`{ print NR, $0 }` printing line numbers (number of records), $0 is entire line

```
{ printf("total pay for %s is $%.2f\n", $1, $2 * $3) }
```

What do these programs do?

`{ print }` print every line

`{ print $0 }` also print every line

`{ print $1, $2 }` print two fields

`{ print $1 $2 }` print one field by concatenating two fields

`{ print NF, $1, $NF }` NF=number of fields, $NF = last field

`{ print $1, $2 * $3 }` compute and print

`{ print NR, $0 }` printing line numbers (number of records), $0 is entire line

`{ printf("total pay for %s is $%.2f\n", $1, $2 * $3) }` – print `$1` as a string `(%s)` and the result of `$2*$3` as a number with 2 digits after the decimal point

AWK Patterns - Selecting Records from the Data File

- By comparison

```
$2 >= 5
```

- By computation

```
$2 * $3 > 50
```

- By text comparison

```
$1 == "Susie"
```

- By combinations of patterns

```
$2 >= 4 || $3 >= 20
```

- By negation

```
! ( $2 < 4 &.&. $3 < 20)
```

- By regular expression enclosed in //

```
$1 ~ /re/
```
(matches)

```
$1 !~ /re/
```
(doesn't match)

Rich Math and String Libraries!!

What do these patterns do?

```
$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
```

AWK 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 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

AWK 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
```

AWK Variables

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

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

```
#print the average of all numbers
awk '{ tot=0; for (i=1; i<=NF; i++) tot += $i; print tot/NF; }'
```

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

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 ^ for exponentiation
- awk uses functions for bit-wise operations: and compl lshift or rshift xor
- Same increment, decrement, and op-assn operators

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
```

String library functions

`length tolower toupper`	Typical string functions
`index`	match Finding substrings1
`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

Arrays

- All arrays are associative
  - Keys can be numbers or strings
  - Vectors can be sparse
  - Indices can be negative
- Uninitialized indices evaluate to 0 or "", depending on context

sparse.awk

```
BEGIN {
a[5] = "spiros"
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 spiros
12 13
13 Ski
-77 I'm here, too
a[7] =
a[7] + 5 = 5
```

Associative Arrays - example

- Remember, all awk arrays are associative
- 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.sample

```
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

Functions – Built in

- Numerical functions
- String functions
- System functions (e.g., call `function` to call any program)

User-Defined 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

User Defined Functions - Example

function myprint(num) { printf "%6.3g\n", num }

function delarray(a, i) { for (i in a) delete a[i] }

```
function myprint(num) {
    printf "%6.3g\n", num
}
```

```
function delarray(a, i) {
    for (i in a)
        delete a[i]
}
```

AWK One Liners

- Like wc

```
$ awk 'END {print NR}'
```

- Like grep

```
$ awk '/regex/'
```

- Like head

```
$ awk 'FNR<=10'
```

- Add line number to front of each record

```
$ awk '{print FNR, $0}'
```

- Print lines 12-23, inclusive

```
$ awk 'FNR==12,FNR==23'
```

- Remove blank lines

```
$ awk '/./'
```

```
$ awk 'NF>0'
```

Lessons

- Lesson 1: AWK does not stand for awkard
- Lesson 2: Awk is a cornerstone for Unix Shell Programming
- Lesson 3: Learn simple AWK and you will do amazing things in the shell

Resources

- These notes
- Unix in a nutshell, 4th Edition (2005) http://tinyurl.com/tahe47t
  - Chapter 11

Acknowledgements

These slides are copied or inspired by the work of the following courses and people:

- CS265 & CS571, Drexel University, Kurt Schmidt, Jeremy Johnson, Geoffrey Mainland, Spiros Mancoridis, Vera available at https://www.cs.drexel.edu/~kschmidt/CS265/