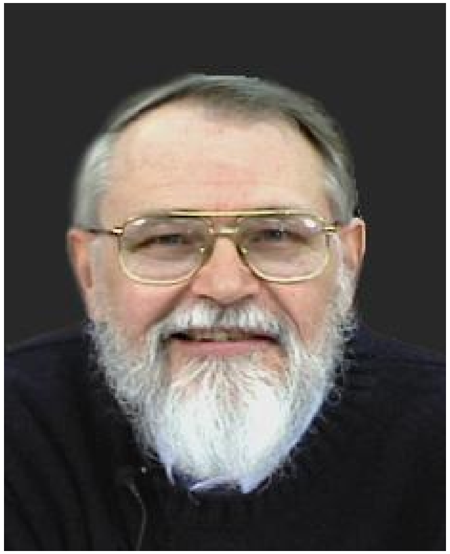
Lecture 5

awk: Programmable Filters

Intro to Shell Scripting

Why is it called AWK?



Aho *Weinberger* *Kernighan*

Awk Introduction

- **awk**: A general purpose programmable filter that handles text (strings) as easily as numbers
  - *This makes **awk** one of the most powerful of the Unix utilities*
- **awk** processes *fields* while **sed** only processes lines
- **nawk** (new **awk**) is the new standard for **awk**
  - *Designed to facilitate large **awk** programs*
  - ***gawk** is a free **nawk** clone from GNU*
- **awk** gets its input from
  - *files*
  - *redirection and pipes*
  - *directly from standard input*

AWK Highlights

- A programming language for handling common data manipulation tasks with only a few lines of code
- **awk** is a *pattern-action* language, like **sed**
- The language looks a little like *C* but automatically handles input, field splitting, initialization, and memory management
  - *Built-in string and numeric data types*
  - *No variable type declarations*
- **awk** is a great prototyping language
  - *Start with a few lines and keep adding until it does what you want*

Awk Features over Sed

- Convenient numeric processing
- Variables and control flow in the actions
- Convenient way of accessing fields within lines
- Flexible printing
- Built-in arithmetic and string functions
- C-like syntax

Structure of an AWK Program

- An **awk** program consists of:
  - *An optional BEGIN segment*
    - *For processing to execute prior to reading input*
  - *pattern - action pairs*
    - *Processing for input data*
    - *For each pattern matched, the corresponding action is taken*
  - *An optional END segment*
    - *Processing after end of input data*

```
BEGIN {action}
pattern {action}
pattern {action}
  .
  .
  .
pattern {action}
END {action}
```

Running an AWK Program

- There are several ways to run an **awk** program:
  - **awk *'program' input_file(s)***
    - *program and input files are provided as command-line arguments*
  - **awk *'program'***
    - *program is a command-line argument; input is taken from standard input (yes, awk is a filter!)*
  - **awk -f *program_file input_files***
    - *program is read from a file*

Patterns and Actions

- Search a set of files or stdin for *patterns*.
- Perform specified *actions* upon lines or fields that contain instances of patterns.
- Does not alter input files.
- Process one input line at a time
- This is similar to **sed**

Pattern-Action Structure

- Every program statement has to have a *pattern* **or** an *action* **or** both
- Default *pattern* is to match all lines
- Default *action* is to print current record
- Patterns are simply listed; actions are enclosed in `{ }`
- **awk** scans a sequence of input *lines*, or *records*, one by one, searching for lines that match the pattern
  - *Meaning of match depends on the pattern*

Patterns

- Selector that determines whether *action* is to be executed
- *pattern* can be:
  - *the special token **BEGIN** or **END***
  - *regular expression (enclosed with //)*
  - *relational or string match expression*
  - ***!** negates the match*
  - *arbitrary combination of the above using* **`&&`** **`||`**
    - **`/NYU/`** *matches if the string "NYU" is in the record*
    - **`x > 0`** *matches if the condition is true*
    - **`/NYU/ && (name == "Open Source Tools")`**

BEGIN and END patterns

- **BEGIN** and **END** provide a way to gain control before and after processing, for initialization and wrap-up.
  - ***BEGIN****: actions are performed before the first input line is read.*
  - ***END****: actions are done after the last input line has been processed.*

Actions

- *action* may include a list of one or more C like statements, as well as arithmetic and string expressions and assignments and multiple output streams.
- *action* is performed on every line that matches *pattern*.
  - *If pattern is not provided, action is performed on every input line*
  - *If action is not provided, all matching lines are sent to standard output.*
- Since *patterns* and *actions* are optional, *actions* must be enclosed in braces to distinguish them from *pattern*.

An Example

```
ls | awk '
BEGIN { print "List of html files:" }
/\.html$/ { print }
END { print "There you go!" }
'
```

```
List of html files:
index.html
as1.html
as2.html
There you go!
```

Variables

- **awk** scripts can define and use variables:

```
BEGIN { sum = 100 }
{ sum ++ }
END { print sum }
```

- Some variables are predefined

Records

- Default record separator is **newline**
  - *By default,* ***awk*** *processes its input a line at a time.*
- Could be any other *regular expression*.
- **RS**: record separator
  - *Can be changed in* ***BEGIN*** *action*
- **NR** is the variable whose value is the number of the current record.

Fields

- Each input line is split into fields.
  - ***FS***: *field separator: default is white space (1 or more spaces or tabs)*
  - ***`awk -F`****ERE option sets* ***FS*** *to the regexp ERE*
    Can also be changed in BEGIN
  - ***$0*** *is the entire line*
  - ***$1*** *is the first field,* ***$2*** *is the second field, ...*
- Only fields begin with **$**, variables are unadorned

Simple Output From AWK

- Printing Every Line
  - If an action has no pattern, the action is performed to all input lines
    - **`{ print }`** prints all input lines to standard out
    - **`{ print $0 }`** does the same thing
- Printing Certain Fields
  - Multiple items can be printed on the same output line with a single print statement
  - **`{ print $1, $3 }`**
  - Expressions separated by a comma are, by default, separated by a single space when printed (**`OFS`**)

Output (continued)

- **NF**, the Number of Fields
  - Any valid expression can be used after a **`$`** to indicate the contents of a particular field
  - One built-in expression is **NF**, or Number of Fields
  - **`{ print NF, $1, $NF }`** will print the number of fields, the first field, and the last field in the current record
  - **`{ print $(NF-2) }`** prints the third to last field
- Computing and Printing
  - You can also do computations on the field values and include the results in your output
  - **`{ print $1, $2 * $3 }`**

Output (continued)

- Printing Line Numbers
  - The built-in variable **NR** can be used to print line numbers
  - **`{ print NR, $0 }`** will print each line prefixed with its line number
- Putting Text in the Output
  - You can also add other text to the output besides what is in the current record
  - **`{ print "total pay for", $1, "is", $2 * $3 }`**
  - Note that the inserted text needs to be surrounded by double quotes

Fancier Output

- Lining Up Fields
  - Like C, **awk** has a printf function for producing formatted output
  - **printf** has the form

    printf(format, val1, val2, val3, ...)

    ```
    { printf("total pay for %s is $%.2f\n",
             $1, $2 * $3) }
    ```
  - When using printf, formatting is under your control so no automatic spaces or newlines are provided by **awk**. You have to insert them yourself.

    ```
    { printf("%-8s %6.2f\n", $1, $2 * $3 ) }
    ```

Selection

- **awk** patterns are good for selecting specific lines from the input for further processing
  - *Selection by Comparison*

    ```
    $2 >= 5 { print }
    ```
  - *Selection by Computation*

    ```
    $2 * $3 > 50 { printf("%6.2f for %s\n",
                          $2 * $3, $1) }
    ```
  - *Selection by Text Content*

    ```
    $1 == "NYU"
    $2 ~ /NYU|CUNY/
    ```
  - *Combinations of Patterns*

    ```
    $2 >= 4 || $3 >= 20
    ```
  - *Selection by Line Number*

    ```
    NR >= 10 && NR <= 20
    ```

Arithmetic and variables

- **awk** variables take on numeric (floating point) or string values according to context.
- User-defined variables need not be declared.
- By default, user-defined variables are initialized to the null string which has numerical value 0.

Computing with AWK

- Counting is easy to do with Awk

```
$3 > 15 { emp = emp + 1}
END { print emp, "employees worked
    more than 15 hrs"}
```

- Computing Sums and Averages is also simple

```
{ pay = pay + $2 * $3 }
END { print NR, "employees"
      print "total pay is", pay
      print "average pay is", pay/NR
    }
```

Handling Text

- One major advantage of **awk** is its ability to handle strings as easily as many languages handle numbers
- **awk** variables can hold strings of characters as well as numbers, and **awk** conveniently translates back and forth as needed
- This program finds the employee who is paid the most per hour:

```
# Fields: employee, payrate
$2 > maxrate { maxrate = $2; maxemp = $1 }
    END { print "highest hourly rate:",
            maxrate, "for", maxemp }
```

String Manipulation

- String Concatenation
  - New strings can be created by combining old ones

```
{ names = names " " $1 }
END { print names }
```

- Printing the Last Input Line
  - Although NR retains its value after the last input line has been read, $0 does not

```
{ last = $0 }
END { print last }
```

Built-in Functions

- **awk** contains a number of built-in functions. length is one of them.
- Counting lines, words, and characters using length (a poor man's **wc**)

```
{
  nc = nc + length($0) + 1
  nw = nw + NF
}
END { print NR, "lines,", nw, "words,", nc,
          "characters" }
```

- **substr(s, m, n)** produces the substring of *s* that begins at position *m* and is at most *n* characters long.

Control Flow Statements

- **awk** provides several control flow statements for making decisions and writing loops
- If-Then-Else

```
$2 > 6 { n = n + 1; pay = pay + $2 * $3 }

END { if (n > 0)
        print n, "employees, total pay is",
              pay, "average pay is", pay/n
      else
        print "no employees are paid more
               than $6/hour"
    }
```

Loop Control

- While

```
# interest1 - compute compound interest
#   input: amount, rate, years
#   output: compound value at end of each year
{ i = 1
  while (i <= $3) {
     printf("\t%.2f\n", $1 * (1 + $2) ^ i)
     i = i + 1
  }
}
```

Do-While Loops

- Do-While

```
do {
    statements
}
while (expression)
```

For statements

- For

```
# interest2 - compute compound interest
#   input: amount, rate, years
#   output: compound value at end of each year

{
  for (i = 1; i <= $3; i = i + 1)
    printf("\t%.2f\n", $1 * (1 + $2) ^ i)
}
```

Arrays

- Array elements are not declared
- Array subscripts can have ***any*** value:
  - *Numbers*
  - *Strings!* ***(associative arrays)***
- Examples

```
arr[3]="value"
grade["Korn"]=40.3
```

Array Example

```
# reverse - print input in reverse order by line

{ line[NR] = $0 }     # remember each line

END {
    for (i=NR; (i > 0); i=i-1) {
        print line[i]
    }
}
```

- Use **`for`** loop to read associative array
  - **`for (v in array) { … }`**
  - Assigns to v each subscript of array (unordered)
  - Element is **`array[v]`**

Awk Variables

- `$0`, `$1`, `$2`, `$NF`
- **NR** - Number of records processed
- **NF** - Number of fields in current record
- **FILENAME** - name of current input file
- **FS** - Field separator, space or TAB by default
- **OFS** - Output field separator, space by default

Useful One (or so)-liners

- `END { print NR }`
- `NR == 10`
- `{ print $NF }`
- ```
  { field = $NF }
  END { print field }
  ```
- `NF > 4`
- `$NF > 4`
- ```
  { nf = nf + NF }
   END { print nf }
  ```

More One-liners

- `/Jeff/ { nlines = nlines + 1 }`
  `END   { print nlines }`
- `$1 > max { max = $1; maxline = $0 }`
  `END      { print max, maxline }`
- `NF > 0`
- `length($0) > 80`
- `{ print NF, $0}`
- `{ print $2, $1 }`
- `{ temp = $1; $1 = $2; $2 = temp; print }`
- `{ $2 = ""; print }`

Even More One-liners

- ```
  { for (i = NF; i > 0; i = i - 1)
        printf("%s ", $i)
    printf("\n")
  }
  ```
- ```
  { sum = 0
    for (i = 1; i <= NF; i = i + 1)
        sum = sum + $i
    print sum
  }
  ```
- ```
  { for (i = 1; i <= NF; i = i + 1)
        sum = sum $i
  }
  END { print sum }
  ```
  ```

Operators

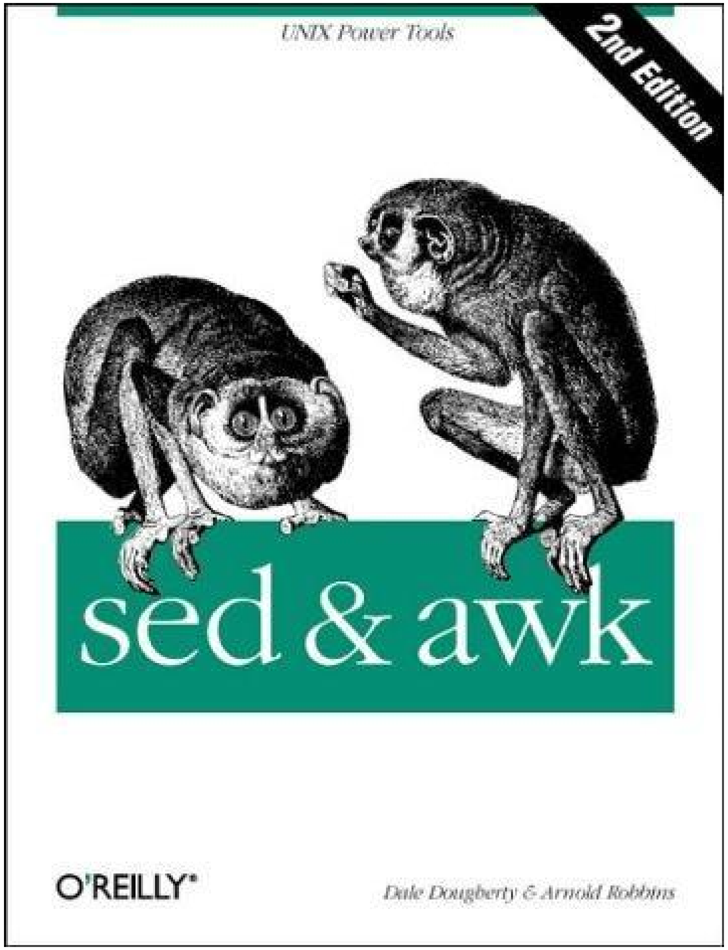
- = assignment operator; sets a variable equal to a value or string
- == equality operator; returns TRUE is both sides are equal
- != inverse equality operator
- && logical AND
- || logical OR
- ! logical NOT
- <, >, <=, >= relational operators
- +, -, /, *, %, ^
- String concatenation

□

Built-In Functions

- Arithmetic
  - ***sin**, **cos**, **atan**, **exp**, **int**, **log**, **rand**, **sqrt***
- String
  - ***length**, **substr, split***
- Output
  - ***print**, **printf***
- Special
  - ***system** - executes a Unix command*
    - *system("clear") to clear the screen*
    - *Note double quotes around the Unix command*
  - ***exit** - stop reading input and go immediately to the END pattern-action pair if it exists, otherwise exit the script*

More Information



on the website

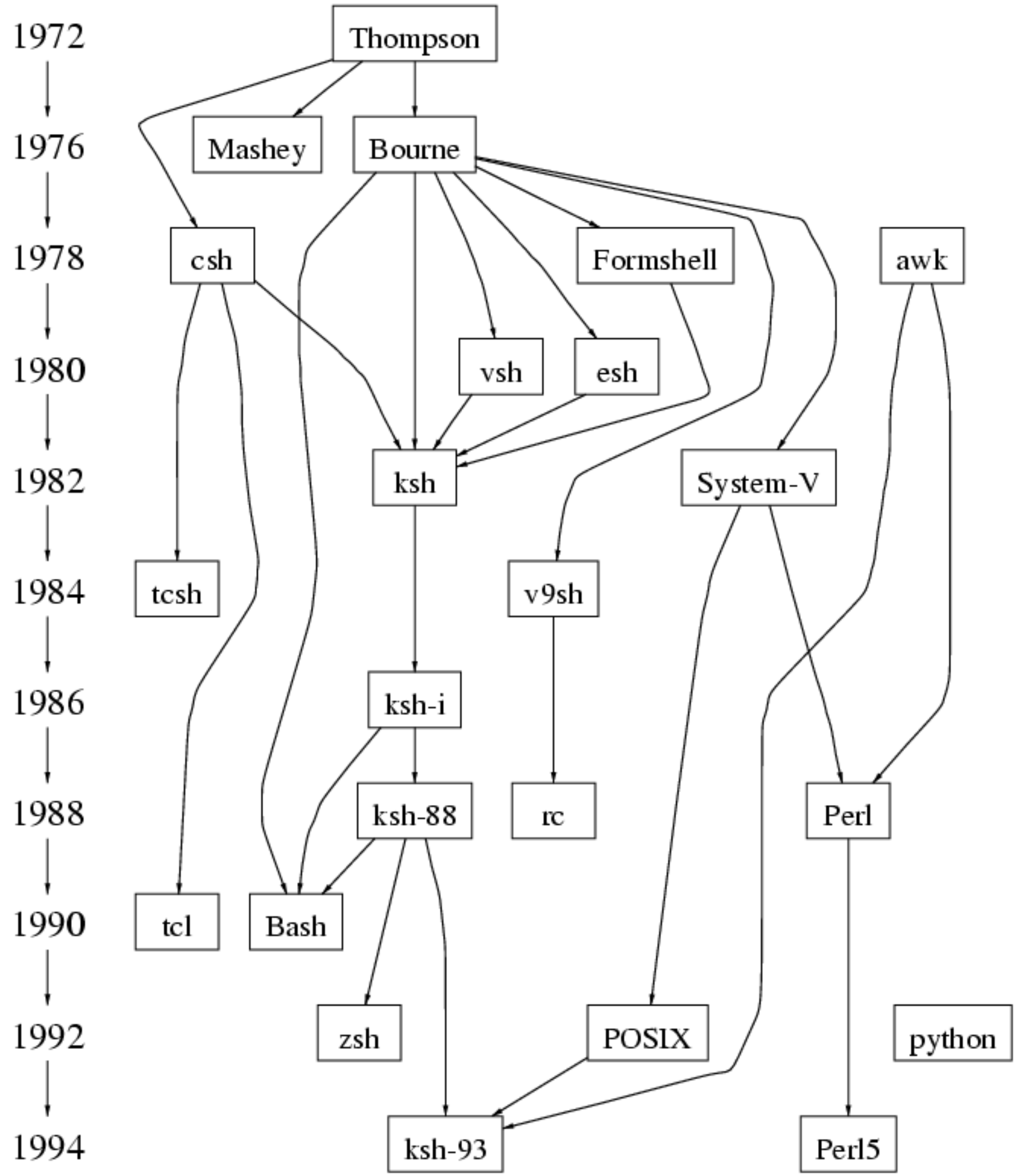
Shell Scripting Intro

What is a shell?

- The user interface to the operating system
- Functionality:
  - Execute other programs
  - Manage files
  - Manage processes
- Full programming language
- A program like any other
  - This is why there are so many shells

Shell History

- There are many choices for shells
- Shell features evolved as UNIX grew



Most Commonly Used Shells

`/bin/csh`	*C shell*
`/bin/tcsh`	*Enhanced C Shell*
`/bin/sh`	*The Bourne Shell / POSIX shell*
`/bin/ksh`	*Korn shell*
`/bin/bash`	*Korn shell clone, from GNU*

Ways to use the shell

Interactively

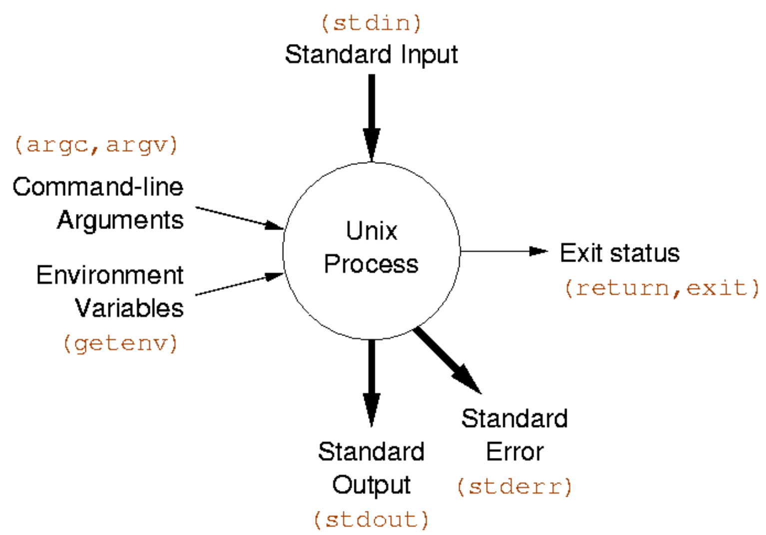
When you log in, you interactively use the shell

Scripting

A set of shell commands that constitute an executable program

UNIX Programs

- **Means of input**:
  - Program arguments [control information]
  - Standard input [data]
  - Environment variables [state information]
- **Means of output**:
  - Return status code [control information]
  - Standard out [data]
  - Standard error [error messages]



Shell Scripts

- A shell script is a regular text file that contains shell or UNIX commands
  - Before running it, it must have execute permission:
    `chmod +x filename`
- A script can be invoked as:
  - **`sh name [ arg … ]`**
  - **`sh < name [ args … ]`**
  - **`name [ arg …]`**

Shell Scripts

- When a script is run, the **kernel** determines which shell it is written for by examining the first line of the script
  - If 1st line starts with **`#!pathname-of-shell`**, then it invokes pathname and sends the script as an argument to be interpreted
  - If **`#!`** is not specified, the current shell assumes it is a script in its own language
    - leads to problems

Simple Example

```
#!/bin/sh

echo Hello World
```

Scripting vs. C Programming

- Advantages of shell scripts
  - *Easy to work with other programs*
  - *Easy to work with files*
  - *Easy to work with strings*
  - *Great for prototyping. No compilation*
- Disadvantages of shell scripts
  - *Slower*
  - *Not well suited for algorithms & data structures*

The C Shell

- C-like syntax (uses `{ }`'s)
- **Inadequate for scripting**
  - Poor control over file descriptors
  - Difficult quoting `"I say \"hello\""` doesn't work
  - Can only trap SIGINT
  - Can't mix flow control and commands
- Survives mostly because of interactive features.
  - Job control
  - Command history
  - Command line editing, with arrow keys (**tcsh**)

http://www.faqs.org/faqs/unix-faq/shell/csh-whynot