SHELL SCRIPTING

Christine Julien

Based on materials from Miryung Kim and Adnan Aziz and from Reva Freedman at Northern Illinois University

But first… the command line

- Things you can do from a command line interface:
  - All of those things we talked about last time
  - Run any program
  - Examples of “programs” that I run from the command line on a daily basis
    - svn
    - ssh
    - javac
    - java
    - gcc

Scripting Languages

- Originally designed as tools for quick hacks, rapid prototyping, and gluing together program
- Evolved into “mainstream” programming languages
- Characteristics
  - Text strings are the basic (sometimes only) data type
  - Associative arrays are a basic aggregate type
  - Regular expressions (regexps) are (usually) built-in
  - There are minimal types and declarations
  - Usually interpreted rather than compiled
  - Easy to get started
- Examples
  - shell, awk, perl, PHP, Ruby, Python, Tcl, Lua, Javascript, Actionscript, VB

AWK

What is awk?

- created by: Aho, Weinberger, and Kernighan
- scripting language used for manipulating data and generating reports

- versions of awk
  - awk, nawk, mawk, pgawk, …
  - GNU awk: gawk

What can you do with awk?

- awk operation:
  - scans a file line by line
  - splits each input line into fields
  - compares input line/fields to pattern
  - performs action(s) on matched lines
- Useful for:
  - transforming data files
  - producing formatted reports
- Programming constructs:
  - formatting output lines
  - arithmetic and string operations
  - conditionals and loops

Basic awk Syntax

- **`awk [options] 'script' file(s)`**

- **`awk [options] –f scriptfile file(s)`**

Options:

-F to change input field separator

-f to name script file

Basic awk Program

- consists of patterns & actions:

```
pattern {action}
```

  - if pattern is missing, action is applied to all lines
  - if action is missing, the matched line is printed
  - must have either pattern or action

Example:

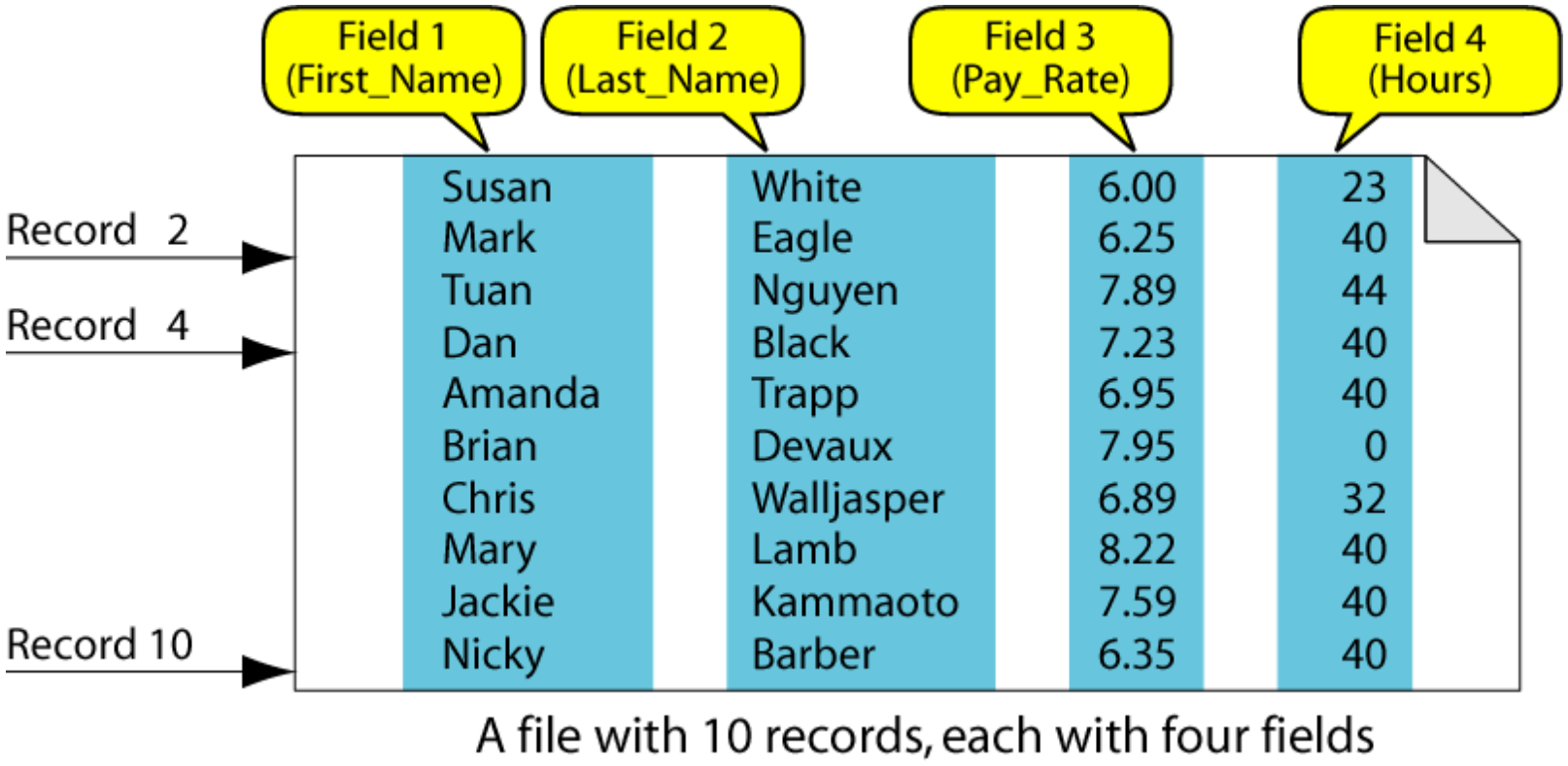
```
awk '/for/' testfile
```

  - prints all lines containing string “for” in testfile

Basic Terminology: input file

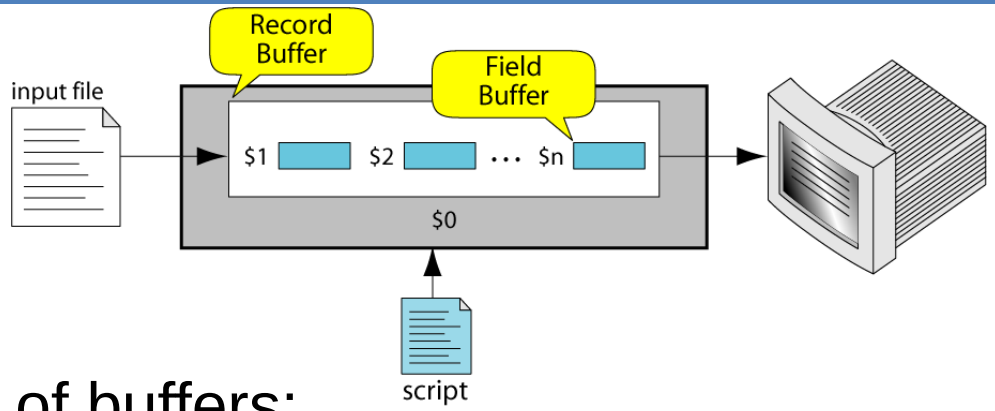
- A **field** is a unit of data in a line
- Each field is separated from the other fields by the **field separator**
  - default field separator is whitespace
- A **record** is the collection of fields in a line
- A data file is made up of records

Example Input File



A file with 10 records, each with four fields

Buffers



- awk supports two types of buffers: record and field
- field buffer:
  - one for each fields in the current record.
  - names: $1, $2, …
- record buffer :
  - $0 holds the entire record

Some System Variables

FS	Field separator (default=whitespace)
RS	Record separator (default=\n)
NF	Number of fields in current record
NR	Number of the current record
OFS	Output field separator (default=space)
ORS	Output record separator (default=\n)
FILENAME	Current filename

Example: Records and Fields

```
% cat emps
Tom Jones          4424     5/12/66 543354
Mary Adams         5346     11/4/63 28765
Sally Chang        1654     7/22/54 650000
Billy Black        1683     9/23/44 336500

% awk '{print NR, $0}' emps
1 Tom Jones        4424     5/12/66 543354
2 Mary Adams       5346     11/4/63 28765
3 Sally Chang      1654     7/22/54 650000
4 Billy Black      1683     9/23/44 336500
```

Example: Space as Field Separator

```
% cat emps
Tom Jones          4424     5/12/66 543354
Mary Adams         5346     11/4/63 28765
Sally Chang        1654     7/22/54 650000
Billy Black        1683     9/23/44 336500

% awk '{print NR, $1, $2, $5}' emps
1 Tom Jones 543354
2 Mary Adams 28765
3 Sally Chang 650000
4 Billy Black 336500
```

Example: Colon as Field Separator

```
% cat em2
Tom Jones:4424:5/12/66:543354
Mary Adams:5346:11/4/63:28765
Sally Chang:1654:7/22/54:650000
Billy Black:1683:9/23/44:336500

% awk -F: '/Jones/{print $1, $2}' em2
Tom Jones 4424
```

Pattern / Action Syntax

```
pattern {statement}
```

(a) One Statement Action

```
pattern {statement1; statement2; statement3}
```

(b) Multiple Statements Separated by Semicolons

```
pattern
{
    statement1
    statement2
    statement3
}
```

(c) Multiple Statements Separated by Newlines

Expression Pattern types

- match
  - entire input record

    regular expression enclosed by ‘/’ s
  - explicit pattern-matching expressions

    ~ (match), !~ (not match)

- expression operators
  - arithmetic
  - relational
  - logical

Example: match input record

```
% cat employees2
Tom Jones:4424:5/12/66:543354
Mary Adams:5346:11/4/63:28765
Sally Chang:1654:7/22/54:650000
Billy Black:1683:9/23/44:336500

% awk -F: '/00$/' employees2
Sally Chang:1654:7/22/54:650000
Billy Black:1683:9/23/44:336500
```

Example: explicit match

```
% cat datafile
northwest NW    Charles Main         3.0    .98    3   34
western   WE    Sharon Gray          5.3    .97    5   23
southwest SW    Lewis Dalsass        2.7    .8     2   18
southern  SO    Suan Chin            5.1    .95    4   15
southeast SE    Patricia Hemenway    4.0    .7     4   17
eastern   EA    TB Savage            4.4    .84    5   20
northeast NE    AM Main              5.1    .94    3   13
north     NO    Margot Weber         4.5    .89    5    9
central   CT    Ann Stephens         5.7    .94    5   13

% awk '$5 ~ /\.[7-9]+/' datafile
southwest SW    Lewis Dalsass        2.7    .8     2   18
central   CT    Ann Stephens         5.7    .94    5   13
```

Examples: matching with REs

```
% awk '$2 !~ /E/{print $1, $2}' datafile
northwest NW
southwest SW
southern SO
north NO
central CT

% awk '/^[ns]/{print $1}' datafile
northwest
southwest
southern
southeast
northeast
north
```

Arithmetic Operators

Operator	Meaning	Example
+	Add	x + y
-	Subtract	x – y
*	Multiply	x * y
/	Divide	x / y
%	Modulus	x % y
^	Exponential	x ^ y

Example:

```
% awk '$3 * $4 > 500 {print $0}' file
```

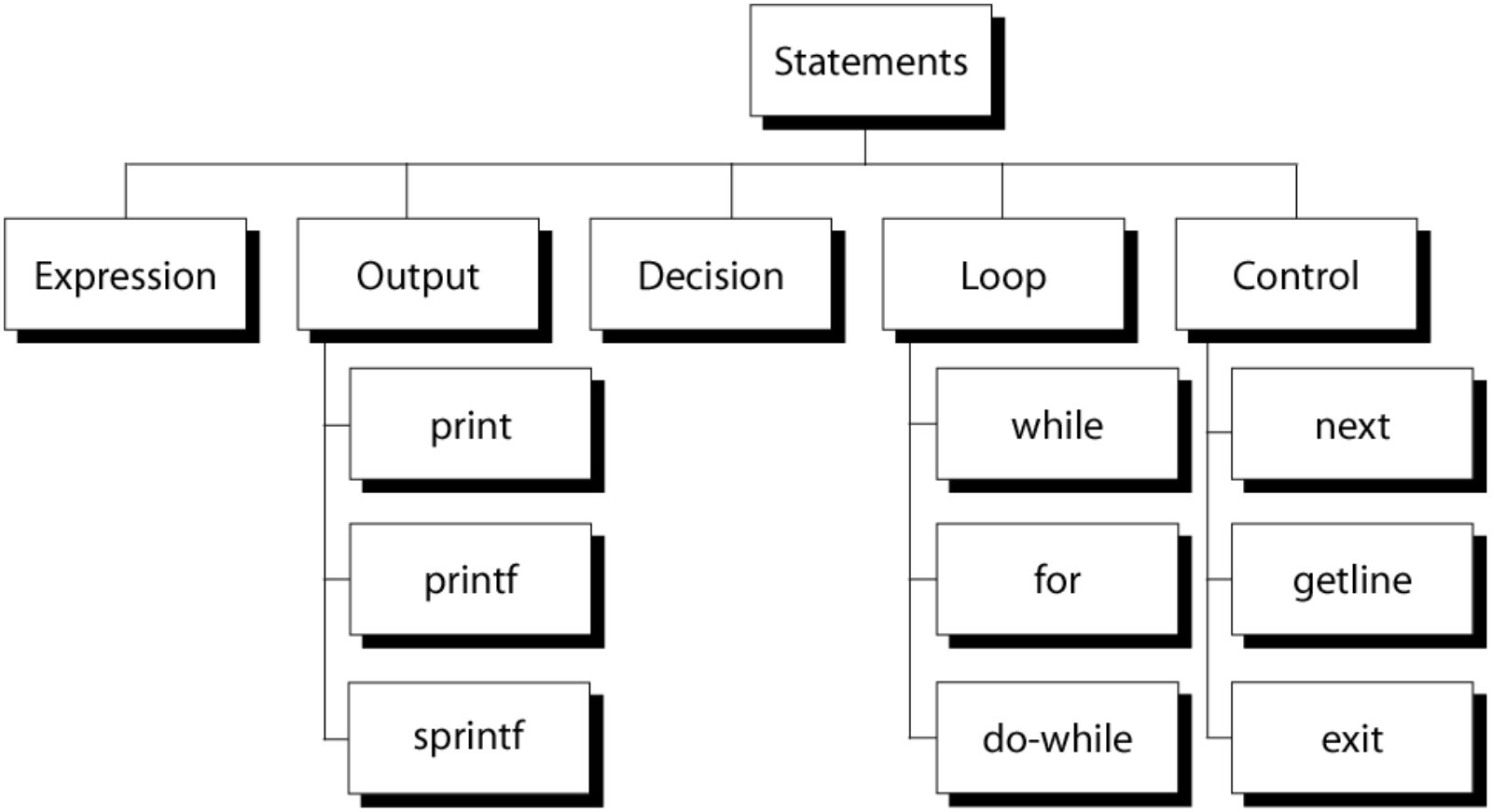
Logical Operators

Operator	Meaning	Example
&&	Logical AND	a && b
||	Logical OR	a || b
!	NOT	! a

Examples:

```
% awk '($2 > 5) && ($2 <= 15)
                        {print $0}' file
% awk '$3 == 100 || $4 > 50' file
```

awk Actions


Statements
Expression
Output
Decision
Loop
Control
print
printf
sprintf
while
for
do-while
next
getline
exit

awk Variables

Format:

```
variable = expression
```

Examples:

```
% awk '$1 ~ /Tom/
          {wage = $3 * $4; print wage}'
        filename
% awk '$4 == "CA"
          {$4 = "California"; print $0}'
        filename
```

Awk example

- File: grades

```
john 85 92 78 94 88
andrea 89 90 75 90 86
jasper 84 88 80 92 84
```

- awk script: average

```
# average five grades
{ total = $2 + $3 + $4 + $5 + $6
  avg = total / 5
  print $1, avg }
```

- Run as:

```
awk -f average grades
```

This is not a class on awk, but…

- You can also make (one-dimensional) arrays
  - Whose index can be a number or string
- There are also various control structures
  - Conditionals (if/else)
  - Repetition
    - For
    - While
    - Do-While

SHELL SCRIPTING

Shell Programming

- The shell is a programming language
- Features include
  - String-valued variables
  - Limited regexps (mostly for filenames)
  - Control-flow
    - If-else (sh syntax)
      - `if cmd; then cmds; elif cmds; else cmds; fi`
    - If-else (csh syntax)
      - `if (expr) cmds; else if (expr) cmds; else cmds; endif`
    - while, for (sh, ksh, bash)
      - `for var in list; do commands; done`
    - for (csh, tcsh)
      - `foreach var (list) commands; end`

Shell Programming

- Shell programming is falling out of favor…
  - Move to GUIs
  - Scripting languages:
    - Sys admin work (e.g., manipulating sets of files, user updates, report generation) is no often done in Perl/Python/Java
- Shell programs are good for personal tools
  - E.g., tailoring the environment or abbreviating common operations (but you can do this with aliases, too)
- Shell programs are good for gluing together existing programs into new ones for prototyping
- Occasionally, shell programs are used for production use, most often for configuration purposes

OH, WAIT…
FILE PERMISSIONS

chmod: Changing File Permissions

- The chmod (change mode) command sets a file's permissions (read, write, and execute) for all three categories of users (owner, group, and others)

command	operation	file
`chmod`	`u+x`	`note`

- The command contains three components:
  - **Category** of user (owner (u), group (g), others (o) or all(a))
  - **Operation** to be performed (add (+), remove (-), or assign (=) a permission)
  - **Permission** type (read (r), write (w), or execute (x))

chmod Examples

- `ls -l` shows the file listing with the permissions

- `chmod u+x test.sh`
  - Adds the executable permission to a file for the user (u)
- `chomd u-rwx test.sh`
  - Removes all permissions from this file for the user
- `chmod a+r,u+w test.sh`
  - Adds the read permission to the fill for all users and the write permission for the user
- `chmod ugo = r test.sh` or `chmod a=r test.sh` or `chmod =r test.sh`
  - Assign the read permission for all users for this file

chmod: The Other Way

- So I can never remember these rules. What I can remember, however are the **masks**.
- Imagine that each of the user, group, and other (in that order) are represented by a three bit mask
  - A “1” in the most significant bit indicates the read permission
  - A “1” in the second most significant bit indicates the write permission
  - A “1” in the least significant bit indicates the execute permission
- So what does this do:

- `chmod 755 test.sh`

BACK TO OUR REGULAR PROGRAMMING…

Shell Variables

- Perform assignment using equals sign without spaces
  - `i=42`
  - `q="What is the answer?"`
- Preface a variable by a dollar sign (`$`) to reference its value
  - `echo $q $i`
  - `a="The answer is $i"`
- Optionally, you can enclose this in braces
  - `a2="The answers are ${i}s"`

Arithmetic

- All values held in variables are strings
  - The shell will treat them as numbers when appropriate (using 0 if necessary)
- There are three ways of performing integer arithmetic
  - `i=`expr $i + 1``
  - `((i=i+1))` or `i=$((i+1))`
  - `let "I = I + 1"`
- Notes:
  - Quotes permit the user of spaces
  - No `$` signs needed with let or inside `((...))`

Loops

```
for variable in list do
    …
done
```

- Lists can be created from
  - The content of an array
  - File pattern
  - Result of a command

Loops: Examples

- What does this do?

```
for i in a b c 1 2 3
do
  echo -n "$i "
done
```

- How about this?

```
count=0
for i in `cat numbers2.txt`
do
  let "c = c + i"
  let "count = count + 1"
done
echo "Number of elements is $count and total is $c"
```

Arrays

- Only one-dimensional arrays
- Arrays do not have “fixed sizes” and can be sparse
- To make an array:
  - `foo=(x y z)`
- To set an element:
  - `foo[2]=hi`
- To get an element:
  - `${foo[2]}`
- To get the number of elements:
  - `${#foo[*]}`
- To get all elements, separated by spaces
  - `${foo[*]}`

You Try It

- Create a script called `arithmetic.sh`
- It should accept some number of integer parameters (at least 2)
- Write an expression using the `expr` notation that adds the first two parameters together
  - Print out the result
- Write an expression using the parentheses method that does some addition and multiplication of the parameters
  - Print out the result
- Write an expression using the let method
  - Print out the result
- Print out all of the provided parameters (use $@)
- Write a loop that computes the sum of the arbitrary number of parameters then prints the sum

What To Submit

- This is Homework 4 on Canvas
- Turn in your `arithmetic.sh` script
  - Be sure to include comments around your actions so the TAs can quickly and easily find and account for each requirement

QUESTIONS?