

Week One, Lecture 2

Agenda

1. Story Time
2. Shell Scripting
3. Reminders
 - Quiz
 - Lab Attendance
 - Homework
 - Office Hours
4. RegEx
5. Group Project Info
 - Application Architecture
 - Methods
6. Slides on sed, awk

The Three Bricklayers



Introduction to Shell Programming

- Shell programming is one of the most powerful features on any UNIX system
- If you cannot find an existing utility to accomplish a task, you can build one using a shell script

Shell Program Structure

- A shell program contains high-level programming language features:
 - Variables for storing data
 - Decision-making control (e.g. if and case statements)
 - Looping abilities (e.g. for and while loops)
 - Function calls for modularity
- A shell program can also contain:
 - UNIX commands
 - Pattern editing utilities (e.g. grep, sed, awk)



Your Shell Programming Library

- Naming of shell programs and their output
 - Give a meaningful name
 - Program name example: `findfile.csh`
 - Do not use: `script1`, `script2`
 - Do not use UNIX command names
- Repository for shell programs
 - If you develop numerous shell programs, place them in a directory (e.g. `bin` or `shellprogs`)
 - Update your path to include the directory name where your shell programs are located

Steps to Create Shell Programs

- Specify shell to execute program
 - Script must begin with `#!` (pronounced “shebang”) to identify shell to be executed

Examples:

```
#! /bin/sh           (defaults to bash, but be explicit)
#! /bin/bash
#! /bin/csh
#! /usr/bin/tcsh
```

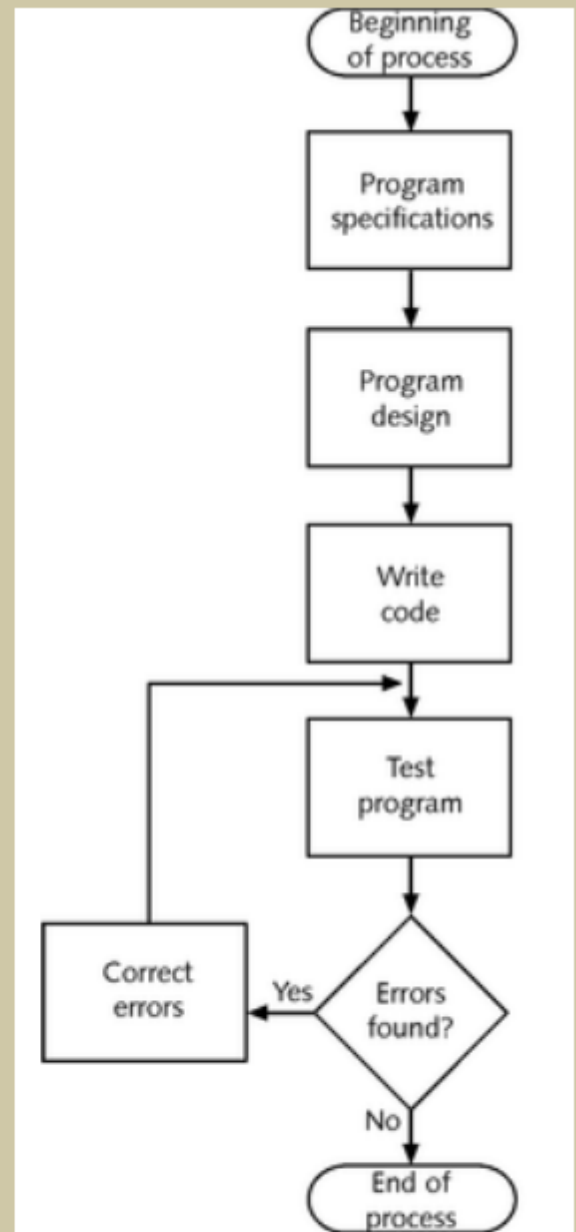
- Make the shell program executable
 - Use the “`chmod`” command to make the program/script file executable

Formatting Shell Programs

- Formatting of shell programs
 - Indent areas (3 or 4 spaces) of programs to indicate that commands are part of a group
 - To break up long lines, place a \ at the end of one line and continue the command on the next line
- Comments
 - Start comment lines with a pound sign (#)
 - Include comments to describe sections of your program
 - Help you understand your program when you look at it later

Steps of Programming

- Guidelines:
 - use good names for
 - script
 - variables
 - use comments
 - lines start with #
 - use indentation to reflect logic and nesting



Example: “HELLO” Script

```
#!/bin/bash
echo "Hello $USER"
echo "This machine is `uname -n`"
echo "The calendar for this month is:"
cal
echo "You are running these processes:"
ps
```

Variables

- 2 types of variables
 - Environment
 - valid for complete login session
 - upper case by convention
 - Shell
 - valid for each shell invocation
 - lower case
- To display value
 echo \$variable

Predefined Shell Variables

Shell Variable	Description
PWD	The most recent current working directory.
OLDPWD	The previous working directory.
BASH	The full path name used of the bash shell.
RANDOM	Generates a random integer between 0 and 32,767
HOSTNAME	The current hostname of the system.
PATH	A list of directories to search of commands.
HOME	The home directory of the current user.
PS1	The primary prompt (also PS2, PS3, PS4).

User-defined Shell Variables

- Syntax:

varname=value

Example:

rate=7.65

echo "Rate today is: \$rate"

- Use double quotes if the value of a variable contains white spaces

Example:

name="Thomas William Flowers"

Numeric variables

- Syntax:

```
let varname=value
```

- Can be used for simple arithmetic:

```
let count=1
```

```
let count=$count+20
```

```
let count+=1
```


Variables commands

- To delete both local and environment variables
unset varname
- To prohibit change
readonly varname
- list all shell variables (including exported)
set



Shell Scripting Tutorials

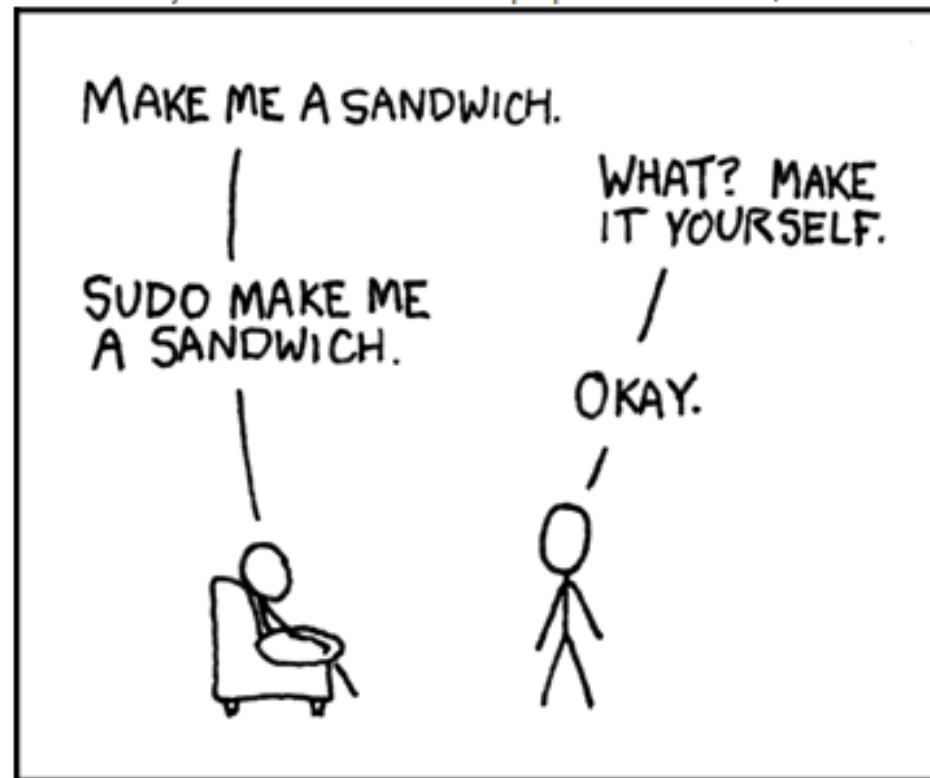
For Practice and Further Reading:

<http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html>

<http://tldp.org/LDP/abs/html>

<https://store.xkcd.com/products/linux-cheat-shirt>

Shift Gears:



A Few Reminders

Reminders on Lab

- Lab Attendance is mandatory
- If you know you must miss, make arrangements to attend another section
- Lab work should be completed during the lab
- If delayed, there is a 20% penalty for late submissions (unless you make other arrangements with your TA)
- Late submissions must be turned in within 24 hours of the end of the lab section
- We are working on grading rubrics for each lab assignment

A Few Reminders

Reminders on Homework

- Homework assignments will appear on the Moodle site at the beginning of the week
- Homework assignments will be due on Wednesday of the NEXT week at 11:59 p.m. – giving you about 10 days to complete the assignment

A Few Reminders

Reminders on Office Hours

Alan Paradise:

Wed & Thu, 3 – 5 p.m., ECOT 520

– By appointment only

Pratima Sherkane:

Sept 6th - Wed - 1 pm to 2 pm, ECOT 831

September 7th - Thurs - 11 am to 12 pm , ECOT 831

Starting September 12th:

Tue - 11 am to 12 pm, ECOT 831

Wed - 1 pm to 2 pm, ECOT 831

Prasanna Srinivasachar:

Tue - 3:00 pm to 4:00 pm at ECOT 831

Thu - 3:00 pm to 4:00 pm at ECOT 832

A Few Reminders

Reminder on Quiz

- My mistake – old version
- Updated this afternoon
- Due by Sunday at 11:59 p.m.

- RegEx: A **regular expression** is a special text string for describing a search pattern.
- Think of **regular expressions** as wildcards on steroids.
- Wildcard notations such as *.txt finds all text files in a directory.
- The regex equivalent is ^.*\.txt\$.

Metacharacters

RE Metacharacter	Matches...
.	Any one character, except new line
[a-z]	Any one of the enclosed characters (e.g. a-z)
*	Zero or more of preceding character
? or \?	Zero or one of the preceding characters
+ or \+	One or more of the preceding characters

- any non-metacharacter matches itself



The grep Utility

- “grep” command:
searches for text in file(s)

Examples:

```
% grep root mail.log
```

```
% grep r..t mail.log
```

```
% grep ro*t mail.log
```

```
% grep 'ro*t' mail.log
```

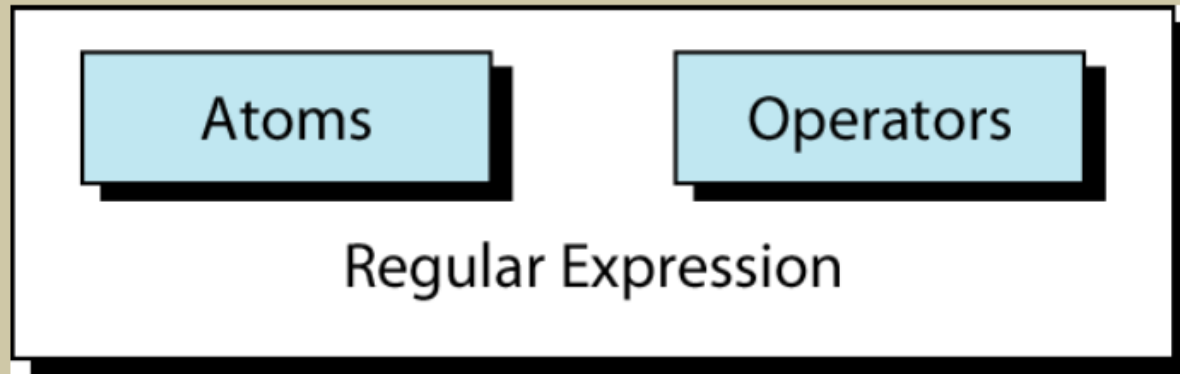
```
% grep 'r[a-z]*t' mail.log
```


more Metacharacters

RE Metacharacter	Matches...
^	beginning of line
\$	end of line
\char	Escape the meaning of <i>char</i> following it
[^]	One character <u>not</u> in the set
\<	Beginning of word anchor
\>	End of word anchor
() or \(\)	Tags matched characters to be used later (max = 9)
 or \ 	Or grouping
x\{m\}	Repetition of character x, m times (x,m = integer)
x\{m,\}	Repetition of character x, at least m times
x\{m,n\}	Repetition of character x between m and m times



Regular Expression

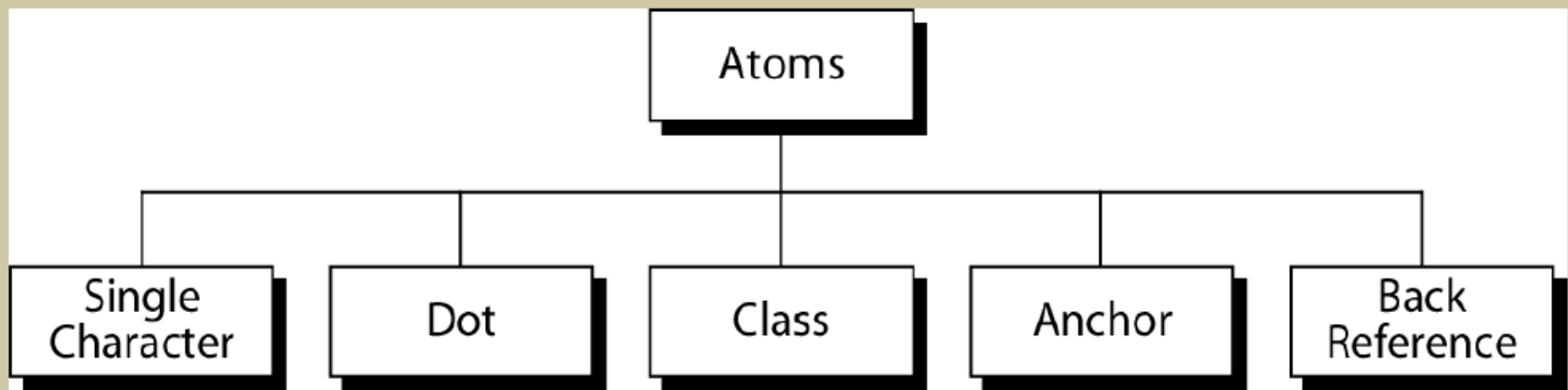


An atom specifies what text is to be matched and where it is to be found.

An operator combines regular expression atoms.

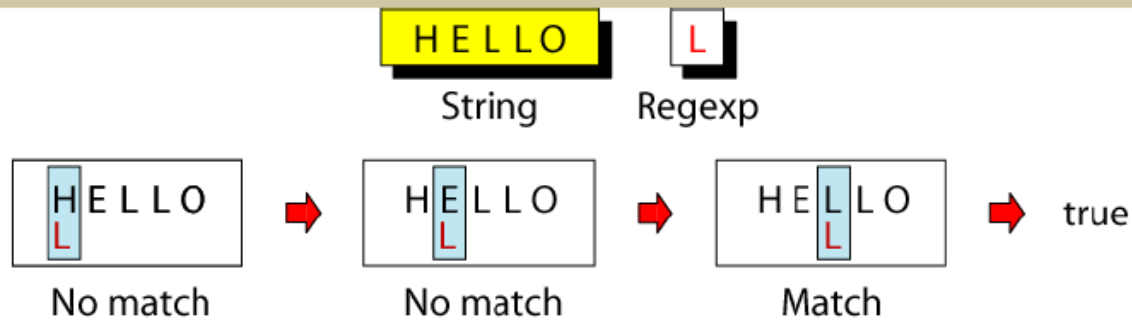
Atoms

An atom specifies what text is to be matched and
where it is to be found.

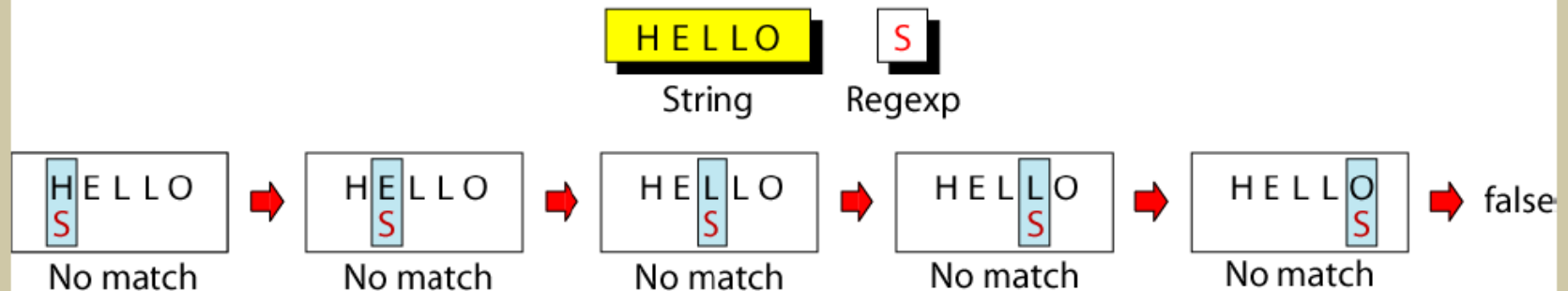


Single-Character Atom

A single character matches itself



(a) Successful Pattern Match

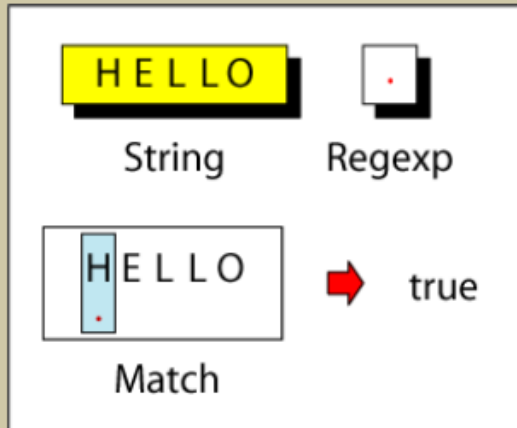


(b) Unsuccessful Pattern Match

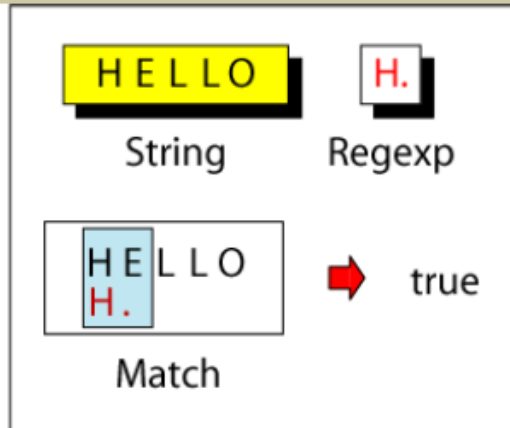


Dot Atom

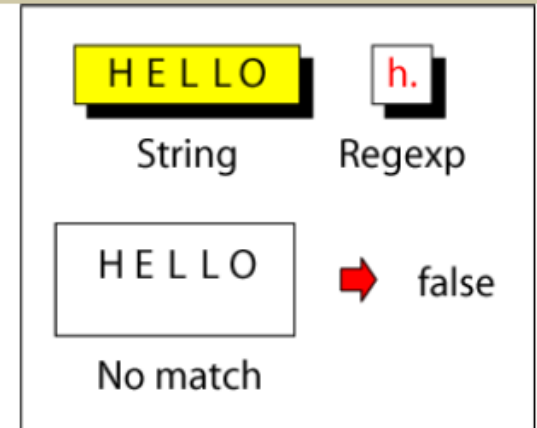
matches **any single character** except for a new line character (`\n`)



(a) Single-Character



(b) Combination-True



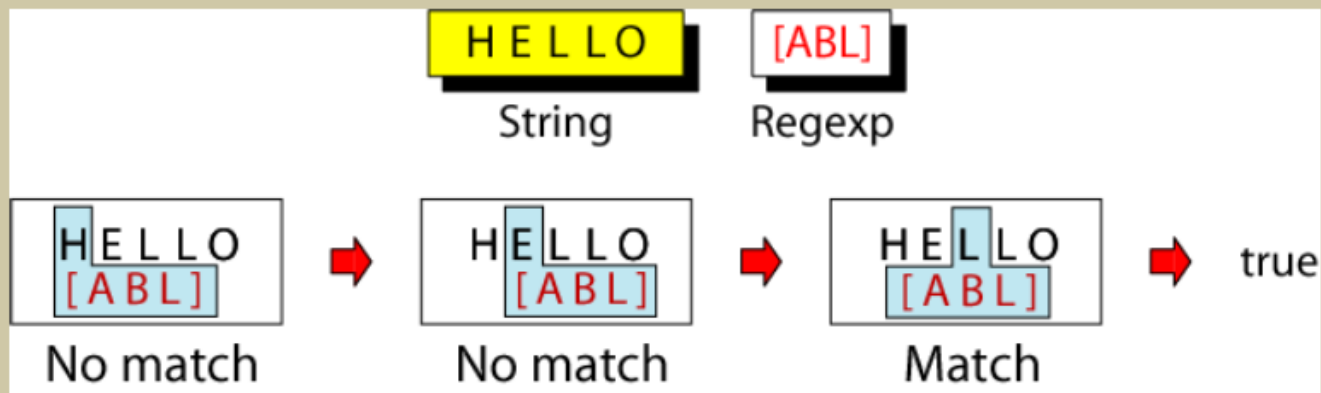
(c) Combination-False



Class Atom

matches only single character that can be any of the characters defined in a set:

Example: [ABC] matches either A, B, or C.











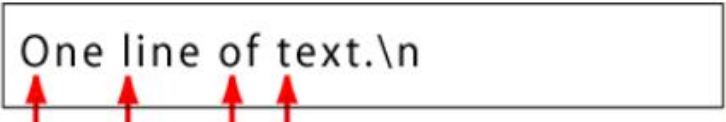



Notes:

- 1) A range of characters is indicated by a dash, e.g. `[A-Q]`
- 2) Can specify characters to be excluded from the set, e.g. `[^0-9]` matches any character other than a number.

RegExpr		Means	RegExpr		Means
[A-H]	➔	[ABCDEFGH]	[^AB]	➔	Any character except A or B
[A-Z]	➔	Any uppercase alphabetic	[A-Za-z]	➔	Any alphabetic
[0-9]	➔	Any digit	[^0-9]	➔	Any character except a digit
[a]	➔	[or a	[]a]	➔] or a
[0-9\ -]	➔	digit or hyphen	[^\^]	➔	Anything except^

Anchors

Anchors tell **where** the next character in the pattern **must** be located in the text data.

Anchor		Means	Example
		Beginning of line	
		End of line	
		Beginning of word	
		End of word	

Sequence Operator

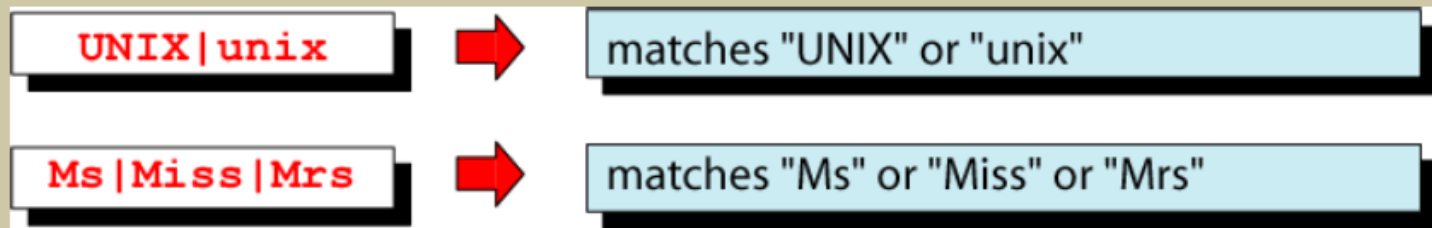
In a sequence operator, if a series of atoms are shown in a regular expression, there is no operator between them.

<code>dog</code>	→	matches the pattern "dog"
<code>a..b</code>	→	matches "a" , any two characters, and "b"
<code>[2-4][0-9]</code>	→	matches a number between 20 and 49
<code>[0-9][0-9]</code>	→	matches any two digits
<code>^\$</code>	→	matches a blank line
<code>^.\$</code>	→	matches a one-character line
<code>[0-9]-[0-9]</code>	→	matches two digits separated by a "-"



Alternation Operator: | or \|

operator (| or \|) is used to define one
or more alternatives



Note: depends on version of “grep”



Repetition Operator: $\{...\}$

The repetition operator specifies that the atom or expression immediately before the repetition may be repeated.

$\{m, n\}$

matches previous character m to n times.

$A\{3, 5\}$



matches "AAA", "AAAA", or "AAAAA"

$BA\{3, 5\}$



matches "BAAA", "BAAAA", or "BAAAAA"



Basic Repetition Forms

Formats

`\{m\}`



matches previous atom exactly m times

`\{m, \}`



matches previous atom m times or more

`\{, n\}`



matches previous atom n times or less

Examples

`CA\{5\}`



CAAAAA

`CA\{3, \}`



CAAA, CAAAA, CAAAAA, ...

`CA\{, 2\}`



C, CA, CAA



Short Form Repetition Operators: * + ?

Formats

*



special case: matches previous atom zero or more times

+



special case: matches previous atom one or more times

?



special case: matches previous atom 0 or one time only

Examples

BA*



B, BA, BAA, BAAA, BAAAA, ...

B.*



B, BA ... BZ, BAA ... BZZ,
BAAA ... BZZZ, ...

.*



zero or more characters

.*



one or more characters

[0-9]?



zero or one digit



Group Operator

In the group operator, when a group of characters is enclosed in parentheses, the next operator applies to the whole group, not only the previous characters.

Regexp		Matches
<code>A(BC)\{3\}</code>	→	<code>ABCBCBC</code>
<code>(F(BC)\{2\}G)\{2\}</code>	→	<code>FBCBCGFBCBCG</code>

Note: depends on version of “grep”
use `\(` and `\)` instead

Further Reading:

- <http://regexone.com/>
- <http://www.zytrax.com/tech/web/regex.htm>
- <http://docs.python.org/2/howto/regex.html>
- <http://misc.yarinareth.net/regex.html>



Objective:

To give students the opportunity to practice using software development methods and tools.

Specifics:

Team formation.

- 5 or 6 students per team
- From the same lab section
- Team composition -- determined by the staff (TA, CAs, & Instructor) based on survey responses.

Requirements

- A working software application
 - includes a “front-end”, middle layer, and a backend database.
- Documentation of the development processes
- Use of git repository for ALL deliverables
- Milestones – evidence of progress along the way
- Each milestone has specific required deliverables
- Work as a team
- Follow a “methodology”

Grading

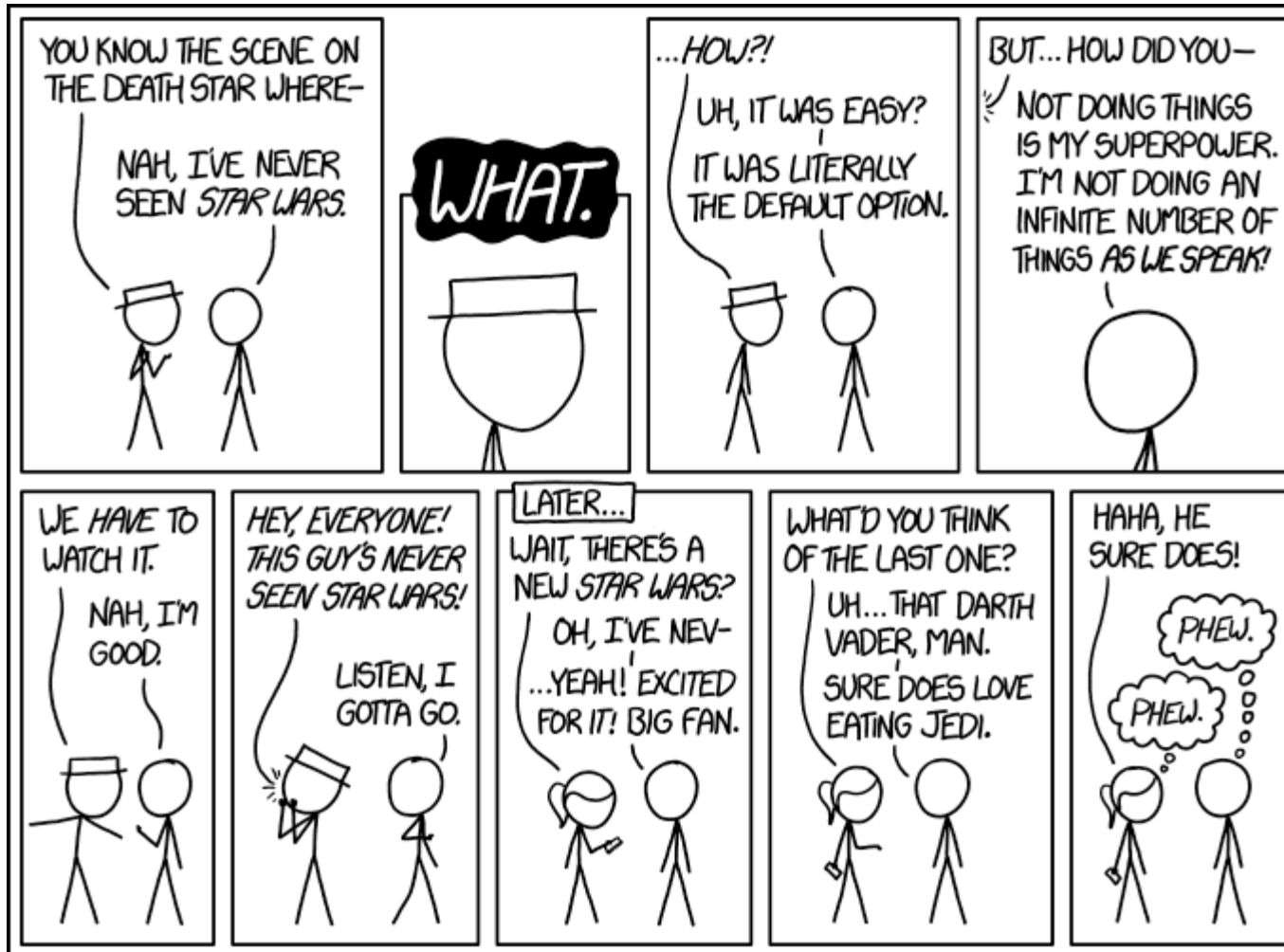
- A working software application
 - includes a “front-end”, middle layer, and a backend database.
- Documentation of the development processes
- Milestones – evidence of progress along the way:
 - Milestones 1-3 and 5-7 are submitted and graded as a team
 - Adjustments based on Participation
 - Milestone # 4 (Individual Interviews)
 - Milestone # 8 (Peer Evaluations)
 - Github “commits”

Group Project Info

Milestones

Milestone 1	40 points	Project Proposal	Week 5
Milestone 2	30 points	Project Tools & Agile Methodology	Week 7
Milestone 3	40 points	Database Design	Week 10
Milestone 4	40 points	Individual Student Meetings and Project Demo	Week 13
Milestone 5	30 points	Unit Testing	Week 14
Milestone 6	40 points	Project Presentations	Week 16
Milestone 7	50 points	Final Submission	Week 17
Milestone 8	10 points	Peer Evaluation and Project Reflection	Week 17

Shift Gears:



AWK

- a programming language designed for text processing
- Used for processing regular expressions in a script
- Used when the text is in file / delimited field format
- typically used as a data extraction and reporting tool
- a powerful standard feature of most Unix-like operating systems.

awk operations:

- 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:

- format output lines for reports
- arithmetic and string operations
- conditionals and loops

Basic awk Script

- 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

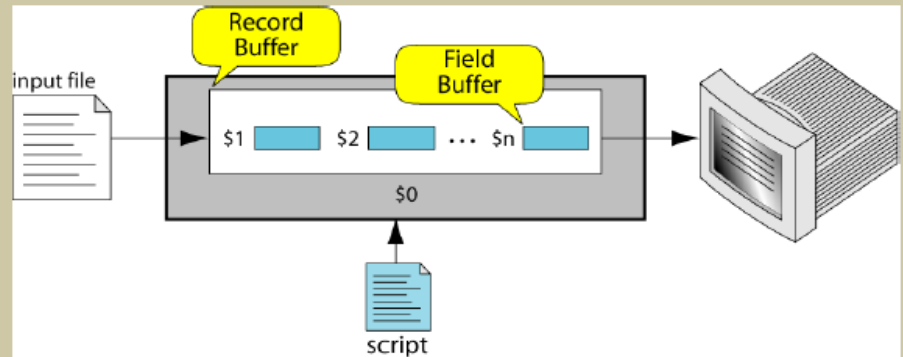
- A field is a unit of data in a line
- Each field is separated from the other fields by the field separator
 - default field separator is whitespace
- A record is the collection of fields in a line
- A data file is made up of records

Example Input File

	Field 1 (First_Name)	Field 2 (Last_Name)	Field 3 (Pay_Rate)	Field 4 (Hours)
Record 2	Susan	White	6.00	23
	Mark	Eagle	6.25	40
Record 4	Tuan	Nguyen	7.89	44
	Dan	Black	7.23	40
	Amanda	Trapp	6.95	40
	Brian	Devaux	7.95	0
	Chris	Walljasper	6.89	32
	Mary	Lamb	8.22	40
Record 10	Jackie	Kammaoto	7.59	40
	Nicky	Barber	6.35	40

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: 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 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
```



- Further Reading:

www.hcs.harvard.edu/~dholland/computers/awk.html

<https://www.digitalocean.com/community/tutorials/how-to-use-the-awk-language-to-manipulate-text-in-linux>

sed

- a programming language designed for text processing
- Used for processing regular expressions in a script
- Used when the text is in a stream (no delimited field structure)
- typically used as a stream editor (thus the name)
- a powerful standard feature of most Unix-like operating systems

sed operations:

- Loops through a file line by line
- Looks for text patterns
- Executes commands on a match
 - Substitute, delete, insert a new line, etc.

Useful for:

- Transforming data files
- Finding text strings and changing them

Programming constructs:

- A rather primitive language

- Further Reading:

<http://www.wikiwand.com/en/Sed>

https://www.tutorialspoint.com/sed/sed_overview.htm