Lecture 4

Regular Expressions

grep and **sed** intro

Previously

- Basic UNIX Commands
  - Files: rm, cp, mv, ls, ln
  - Processes: ps, kill
- Unix Filters
  - **cat**, **head**, **tail**, tee, **wc**
  - **cut**, paste
  - **find**
  - **sort**, **uniq**
  - **comm**, diff, cmp
  - tr

Subtleties of commands

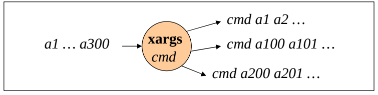
- Executing commands with find
- Specification of columns in cut
- Specification of columns in sort
- Methods of input
  - Standard in
  - File name arguments
  - Special "-" filename
- Options for uniq

Today

- Regular Expressions
  - Allow you to search for text in files
  - **grep** command
- Stream *manipulation*:
  - **sed**
- But first, a command we didn't cover last time…

xargs

- Unix limits the size of arguments and environment that can be passed down to child
- What happens when we have a list of 10,000 files to send to a command?
- **xargs** solves this problem
  - Reads arguments as standard input
  - Sends them to commands that take file lists
  - May invoke program several times depending on size of arguments



a1 … a300

xargs *cmd*

cmd a1 a2 …

cmd a100 a101 …

cmd a200 a201 …

find utility and xargs

- `find . -type f -print | xargs wc -l`
  - `-type f` for files
  - `-print` to print them out
  - **xargs** invokes **wc** 1 or more times

-
  ```
  wc -l a b c d e f g
  wc -l h i j k l m n o
  …
  ```
- Compare to: `find . -type f –exec wc -l {} \;`

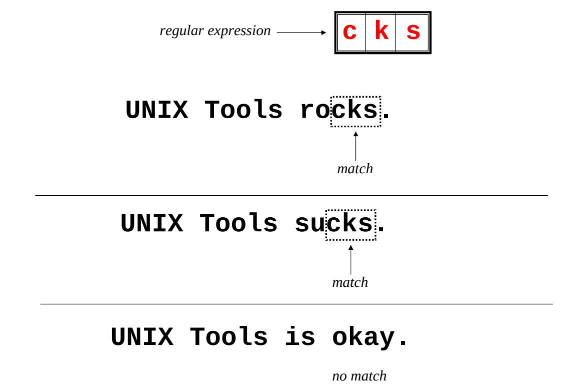
Regular Expressions

What Is a Regular Expression?

- A regular expression (*regex*) describes a set of possible input strings.
- *Regular expressions* descend from a fundamental concept in Computer Science called *finite automata* theory
- *Regular expressions* are endemic to Unix
  - **vi**, **ed**, **sed**, and **emacs**
  - **awk**, **tcl**, **perl** and **Python**
  - **grep**, **egrep**, **fgrep**
  - **compilers**

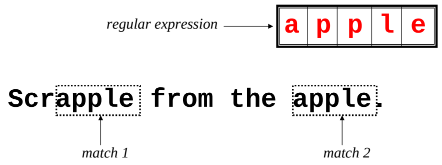
Regular Expressions

- The simplest regular expressions are a string of literal characters to match.
- The string ***matches*** the regular expression if it contains the substring.



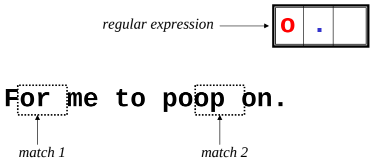
Regular Expressions

- A regular expression can match a string in more than one place.



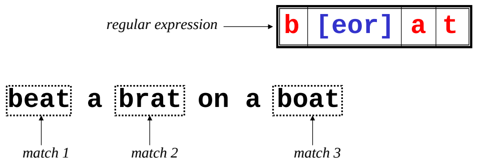
Regular Expressions

- The . regular expression can be used to match any character.



Character Classes

- Character classes `[]` can be used to match any specific set of characters.



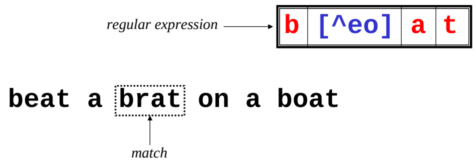
regular expression → `b [eor] a t`

`beat a brat on a boat`

match 1 *match 2* *match 3*

Negated Character Classes

- Character classes can be negated with the `[^]` syntax.



regular expression → `b [^eo] a t`

`beat a brat on a boat`

match

More About Character Classes

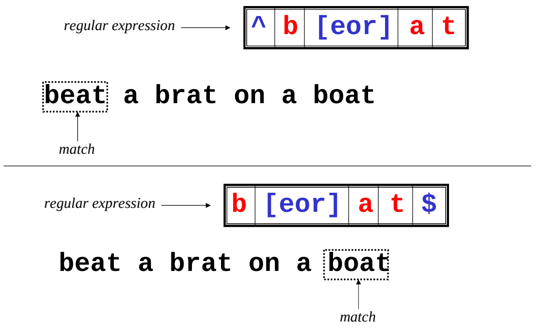
  - `[aeiou]` will match any of the characters `a`, `e`, `i`, `o`, or `u`
  - `[kK]orn` will match `korn` or `Korn`
- Ranges can also be specified in character classes
  - `[1-9]` is the same as `[123456789]`
  - `[abcde]` is equivalent to `[a-e]`
  - You can also combine multiple ranges
    - `[abcde123456789]` is equivalent to `[a-e1-9]`
  - Note that the `-` character has a special meaning in a character class ***but only*** if it is used within a range, `[-123]` would match the characters `-`, `1`, `2`, or `3`

Named Character Classes

- Commonly used character classes can be referred to by name (*alpha*, *lower*, *upper*, *alnum*, *digit*, *punct*, *cntrl*)
- Syntax `[:`***name***`:]`
  - `[a-zA-Z]` `[[:alpha:]]`
  - `[a-zA-Z0-9]` `[[:alnum:]]`
  - `[45a-z]` `[45[:lower:]]`
- Important for portability across languages

Anchors

- Anchors are used to match at the beginning or end of a line (or both).
- `^` means beginning of the line
- `$` means end of the line



regular expression → `^ b [eor] a t`

`beat a brat on a boat`

match

regular expression → `b [eor] a t $`

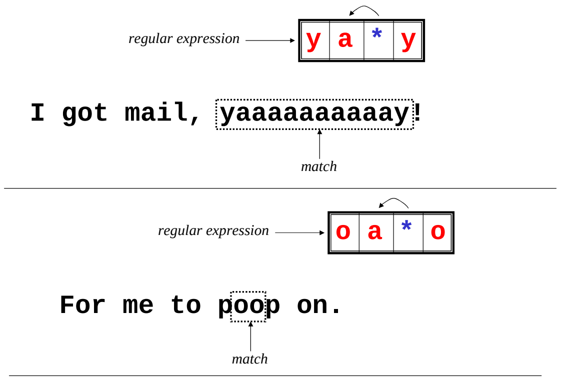
`beat a brat on a boat`

match

`^word$` `^$`

Repetition

- The `*` is used to define **zero or more** occurrences of the *single* regular expression preceding it.



regular expression → `ya*y`

`I got mail, yaaaaaaaaaay!`

match

regular expression → `oa*o`

`For me to poop on.`

match

`.*`

Repetition Ranges

- Ranges can also be specified
  - `{ }` notation can specify a range of repetitions for the immediately preceding regex
  - `{`*`n`*`}` means exactly *n* occurrences
  - `{`*`n`*`,}` means at least *n* occurrences
  - `{`*`n`*`,`*`m`*`}` means at least *n* occurrences but no more than *m* occurrences
- Example:
  - `.{0,}` same as `.*`
  - `a{2,}` same as `aaa*`

Subexpressions

- If you want to group part of an expression so that `*` or `{ }` applies to more than just the previous character, use `( )` notation
- Subexpresssions are treated like a single character
  - `a*` matches 0 or more occurrences of `a`
  - `abc*` matches `ab`, `abc`, `abcc`, `abccc`, …
  - `(abc)*` matches `abc`, `abcabc`, `abcabcabc`, …
  - `(abc){2,3}` matches `abcabc` or `abcabcabc`

grep

- grep comes from the **ed** (Unix text editor) search command “**g**lobal **r**egular **e**xpression **p**rint” or g/re/p
- This was such a useful command that it was written as a standalone utility
- There are two other variants, ***egrep*** and ***fgrep*** that comprise the *grep* family
- *grep* is the answer to the moments where you know you want the file that contains a specific phrase but you can’t remember its name

Family Differences

- **grep** - uses regular expressions for pattern matching
- **fgrep** - file grep, does not use regular expressions, only matches fixed strings but can get search strings from a file
- **egrep** - extended grep, uses a more powerful set of regular expressions but does not support backreferencing, generally the fastest member of the grep family
- **agrep** – approximate grep; not standard

Syntax

- Regular expression concepts we have seen so far are common to **grep** and **egrep**.
- grep and egrep have different syntax
  - **grep**: BREs
  - **egrep**: EREs (enhanced features we will discuss)
- Major syntax differences:
  - **grep**: `\(` and `\)`, `\{` and `\}`
  - **egrep**: `(` and `)`, `{` and `}`

Protecting Regex Metacharacters

- Since many of the special characters used in regexs also have special meaning to the shell, it's a good idea to get in the habit of single quoting your regexs
  - This will protect any special characters from being operated on by the shell
  - If you habitually do it, you won't have to worry about when it is necessary

Escaping Special Characters

- Even though we are single quoting our regexs so the shell won't interpret the special characters, some characters are special **to grep** (eg `*` and `.`)
- To get literal characters, we *escape* the character with a \ (backslash)
- Suppose we want to search for the character sequence 'a*b*'
  - Unless we do something special, this will match zero or more 'a's followed by zero or more 'b's, *not what we want*
  - **'a*b*'** will fix this - now the asterisks are treated as regular characters

Egrep: Alternation

- Regex also provides an alternation character `|` for matching one or another subexpression
  - `(T|Fl)an` will match 'Tan' or 'Flan'
  - `^(From|Subject):` will match the From and Subject lines of a typical email message
    - It matches a beginning of line followed by either the characters 'From' or 'Subject' followed by a ':'
- Subexpressions are used to limit the scope of the alternation
  - `At(ten|nine)tion` then matches "Attention" or "Atnination", not "Atten" or "ninetion" as would happen without the parenthesis - `Atten|ninetion`

Egrep: Repetition Shorthands

- The `*` (star) has already been seen to specify zero or more occurrences of the immediately preceding character
- `+` (plus) means "one or more"
  - `abc+d` will match 'abcd', 'abccd', or 'abccccccd' but will not match 'abd'
  - Equivalent to `{1,}`

Egrep: Repetition Shorthands cont

- The '`?`' (question mark) specifies an optional character, the single character that immediately precedes it
  - `July?` will match 'Jul' or 'July'
  - Equivalent to `{0,1}`
  - Also equivalent to `(Jul|July)`
- The `*`, `?`, and `+` are known as *quantifiers* because they specify the quantity of a match
- Quantifiers can also be used with subexpressions
  - `(a*c)+` will match 'c', 'ac', 'aac' or 'aacaacac' but will not match 'a' or a blank line

Grep: Backreferences

- Sometimes it is handy to be able to refer to a match that was made earlier in a regex
- This is done using *backreferences*
  - **`\n`** is the backreference specifier, where *n* is a number
- Looks for *n*th subexpression
- For example, to find if the first word of a line is the same as the last:
  - **`^\([[:alpha:]]\{1,\}\) .* \1$`**
  - The **`\([[:alpha:]]\{1,\}\)`** matches 1 or more letters

Practical Regex Examples

- Variable names in C
  - **`[a-zA-Z_][a-zA-Z_0-9]*`**
- Dollar amount with optional cents
  - **`\$[0-9]+(\.[0-9][0-9])?`**
- Time of day
  - **`(1[012]|[1-9]):[0-5][0-9] (am|pm)`**
- HTML headers <h1> <H1> <h2> …
  - **`<[hH][1-4]>`**

grep Family

- Syntax

  grep [-hilnv] [-e expression] [filename]

  egrep [-hilnv] [-e expression] [-f filename] [expression] [filename]

  fgrep [-hilnxv] [-e string] [-f filename] [string] [filename]
  - **-h** Do not display filenames
  - **-i** Ignore case
  - **-l** List only filenames containing matching lines
  - **-n** Precede each matching line with its line number
  - **-v** Negate matches
  - **-x** Match whole line only (*fgrep* only)
  - **-e** *expression* Specify expression as option
  - **-f** *filename* Take the regular expression (egrep) or a list of strings (fgrep) from *filename*

grep Examples

- **`grep 'men' GrepMe`**
- **`grep 'fo*' GrepMe`**
- **`egrep 'fo+' GrepMe`**
- **`egrep -n '[Tt]he' GrepMe`**
- **`fgrep 'The' GrepMe`**
- **`egrep 'NC+[0-9]*A?' GrepMe`**
- **`fgrep -f expfile GrepMe`**
- Find all lines with signed numbers

```
$ egrep ’[-+][0-9]+\.?[0-9]*’ *.c
bsearch. c: return -1;
compile. c: strchr("+1-2*3", t-> op)[1] - ’0’, dst,
convert. c: Print integers in a given base 2-16 (default 10)
convert. c: sscanf( argv[ i+1], "% d", &base);
strcmp. c: return -1;
strcmp. c: return +1;
```

- **egrep** has its limits: For example, it cannot match all lines that contain a number divisible by 7.

Fun with the Dictionary

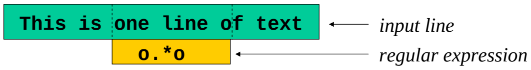
- `/usr/dict/words` contains about 25,000 words
  - **`egrep hh /usr/dict/words`**
    - beachhead
    - highhanded
    - withheld
    - withhold
- **egrep** as a simple spelling checker: Specify plausible alternatives you know

```
egrep "n(ie|ei)ther" /usr/dict/words
neither
```

- How many words have 3 a’s one letter apart?
  - **`egrep a.a.a /usr/dict/words | wc –l`**
    - 54
  - **`egrep u.u.u /usr/dict/words`**
    - cumulus

Other Notes

- Use **`/dev/null`** as an extra file name
  - Will print the name of the file that matched
    - **`grep test bigfile`**
      - **`This is a test.`**
    - **`grep test /dev/null bigfile`**
      - **`bigfile:This is a test.`**
- Return code of grep is useful
  - **`grep fred filename > /dev/null && rm filename`**



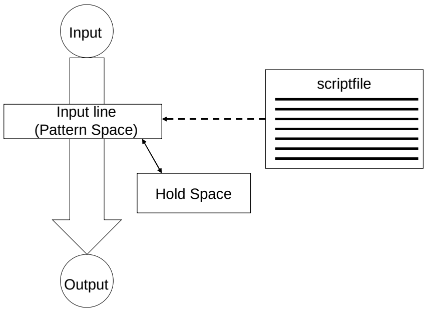
x xyz	Ordinary characters match themselves (NEWLINES and metacharacters excluded) Ordinary strings match themselves	*fgrep, grep, egrep*
\m ^ $. [xy^$x] [^xy^$z] [a-z] r* r1r2	Matches literal character *m* Start of line End of line Any single character Any of x, y, ^, $, or z Any one character other than x, y, ^, $, or z Any single character in given range zero or more occurrences of regex r Matches r1 followed by r2	*grep, egrep*
\(r\) \n \{n,m\}	Tagged regular expression, matches r Set to what matched the *n*th tagged expression (n = 1-9) Repetition	*grep*
r+ r? r1|r2 (r1|r2)r3 (r1|r2)* {n,m}	One or more occurrences of r Zero or one occurrences of r Either r1 or r2 Either r1r3 or r2r3 Zero or more occurrences of r1|r2, e.g., r1, r1r1, r2r1, r1r1r2r1,…) Repetition	*egrep*

Quick Reference

Sed: Stream-oriented, Non-Interactive, Text Editor

- Look for patterns one line at a time, like **grep**
- *Change* lines of the file
- Non-interactive text editor
  - Editing commands come in as *script*
  - There is an interactive editor *ed* which accepts the same commands
- A Unix filter
  - Superset of previously mentioned tools

Sed Architecture



Conceptual overview

- All editing commands in a **sed** script are applied in order to each input line.
- If a command changes the input, subsequent command address will be applied to the current (modified) line in the pattern space, not the original input line.
- The original input file is unchanged (sed is a filter), and the results are sent to standard output (but can be redirected to a file).

Scripts

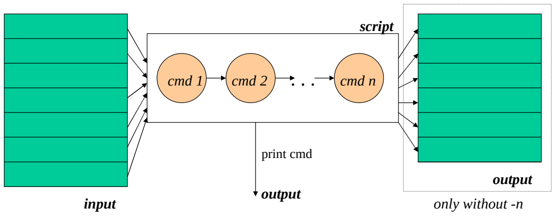
- A script is nothing more than a file of commands
- Each command consists of up to two *addresses* and an *action*, where the *address* can be a regular expression or line number.

address	*action*	*command*
address	*action*	
address	*action*	
address	*action*	
address	*action*	

script

Sed Flow of Control

- *sed* then reads the next line in the input file and restarts from the beginning of the script file
- All commands in the script file are compared to, and potentially act on, all lines in the input file


script
cmd 1
cmd 2
. . .
cmd n
print cmd
input
output
output
only without -n