Overview of character processing

- tr – basic character substitution – **tr**anslate
  - As an introduction to stream editing…
  - Stream!? … a stream of characters.. e.g. a file, pipe between processes etc.,
  - as distinct from interactive online editing…
  - Stream editing can be programmed and run in batch (non-interactive) mode … suited for large jobs…
- Regexp – regular expressions – way to specify patters
  - Ubiquitous throughout CS… used everywhere, in
    interactive editors – ed, vi, vim, emacs … and derivatives…(incl Microsoft) etc.
    stream editors – sed
    Command arguments …e.g. ls *.txt
    SQL queries
    Language specification …although (E)BNF is used to specify grammars
- Grep family … a great find!? – actually find generally uses grep
  - "**g**lobal **r**egular **e**xpression **p**rint" – but find is more meaningful!

TRanslate – tr – i.e. character translate

- The simplest, so check if it will solve problem first
- Copies standard input to standard output with substitution or deletion of selected characters
- Syntax: *tr [-cds] [string1] [string2]*
  - *-c* - complements (everything except) the characters in *string1* with respect to the entire ASCII character set ..
  - *-d* - delete all input characters contained in *string1*
  - *-s* - squeeze all strings of repeated output characters that are in *string2* to single characters
- *tr* provides only simple text processing.
  - It does not allow the full power of *regular expressions*.
  - If you need the power of *regular expressions*, use *sed*

Examples

- *tr* reads from standard input.
  - Any character that does not match a character in *string1* is passed to *standard output* unchanged
  - Any character that does match a character in *string1* is translated into the corresponding character in *string2* and then passed to *standard output*
- Examples
  - *tr s z* replaces all instances of *s* with z
  - *tr so zx* replaces all instances of *s* with *z* and *o* with *x*
  - *tr a-z A-Z* replaces all lower case characters with upper case characters

Odd chars, Space & delimiter replace

- *tr* does a character by character 1-to-1 case sensitive swap
- Normal input output redirection and piping apply.
- This implies that certain characters must be protected from the shell by quotes or \, such as:
  - spaces : ; & ( ) | ^ < > [ ] \ ! newline TAB
- Example
  - *tr o* ‘ ’ replaces all o's with a blank (space) typed but invisible
  - tr ‘ ” ’ “ ‘ “ replace all double quotes with single ones
  - tr \” \’

Octal codes... for some chars

- Non-printing characters can also be specified
  - Bell ‘\07’
  - Backspace ‘\010’
  - CR ‘\015’
  - Escape ‘\033’
  - Formfeed ‘\014’
  - Newline ‘\012’
  - TAB ‘\011’

Codes can also be expressed other bases...
e.g. hexadecimal.

Alternative to Octal codes... for some chars

\NNN character with octal value NNN (1 to 3 octal digits)

\\	backslash
\a	audible BEL
\b	backspace
\f	form feed
\n	new line
\r	return
\t	horizontal tab
\v	vertical tab

Newer character classes...

[:alpha:] *all letters*
[:upper:] *all upper case letters*
[:lower:] *all lower case letters*
[:digit:] *all digits*
[:alnum:] *all letters and digits*
[:graph:] *all printable characters, excluding space*
[:print:] *all printable characters, including space*
[:punct:] *all punctuation characters*
[:blank:] *all horizontal whitespace*
[:space:] *all horizontal or vertical whitespace*
[:cntrl:] *all control characters*
[:xdigit:] *all hexadecimal digits*

- *string1* and *string2* can use ranges of characters as follows:
  - *tr a-z A-Z* translates all lower case to upper case
  - *tr a-m A-M* translates only
    lower case a through m to upper case A though M
- Ranges must be in ascending ASCII order
- What would happen if:
  tr 1-9 9-1

- *string1* and *string2* can use ranges of characters as follows:
  - *tr a-z A-Z* translates all lower case to upper case
  - *tr a-m A-M* translates only
    lower case a through m to upper case A though M
- Ranges must be in ascending ASCII order
- What would happen if:
  tr 1-9 9-1

- *tr* would interpret 9-1 as three individual characters since they are not in ascending ASCII order
  so if the numbers 1 through 9 were entered,
  tr would output 9, -, 1, 1, 1……...

- The *tr -d* option lets you delete any character matched in *string1*. *string2* is not allowed with the *-d* option
- Examples
  - *tr -d a-z* deletes all lower case characters
  - *tr -d aeiou* deletes all vowels
  - *tr -dc aeiou* deletes all character **except** vowels
    (note: this includes spaces, TABS, and newlines as well)

- Normal usage would be with pipes or redirection
  tr -d '\015' <in_file >out_file
  - Converts a DOS text file to a Unix text file by removing CR
    (DOS uses CR + newline/linefeed, Unix just uses newline/linefeed)

(Reverse is not a deletion or 1-1 char map, so done with sed instead)

DOS 2 IX : newer char class

- Newer char class
  $ tr -d \\r < dosfile.txt > unixfile.txt
  the first '\' escapes the second '\' ;
  so taken literally as '\r', CR – carriage return
  (see a few slides back)

- Reverse IX 2 DOS
  sed s/$/"\r"/ dosfile.txt > unixfile.txt
  - since tr does only 1 to 1, it cannot do 1 to 2,
    so cannot append CR to NL or LF (newline/linefeed)
  - quotes requirement depends on installation & configuration

DOS 2 IX : inbuilt MS ↔ IX file conv.

- Older
  - dos2unix & unix2dos

- Newer replaces older
  - flip -u filename(s)
    - -u : to IX
    - -m : to MS
    will ensure that all are converted to target format, irrespective of initial format.
  - Aliases: toix & toms

Alternative uses...

- IoT / device monitoring
  - Delete and squeeze repeated ‘ack’ chars
- Tidy up files from script session recording, using char classes as appropriate:
  - *[:blank:]* *all horizontal whitespace*
  - *[:space:]* *all horizontal or vertical whitespace*
  - *[:cntrl:]* *all control characters*

- Example from the tr man page. How does this work?

*tr -cs A-Za-z ‘**0**012’ <in_file > out_file*

- It replaces all characters that
  - are not letters (-c for complement) with a newline (\012)
  - then squeezes multiple newlines into a single newline (-s)

More tr Examples

- The following commands are equivalent
  - tr ‘abcdef’ ‘xyzabc’
  - tr ‘[a-c][d-f]’ ‘[x-z][a-c]’
- This command implements the “rotate 13” encryption
  - tr ‘[A-M][N-Z][a-m][n-z]’ ‘[N-Z][A-M][n-z][a-m]’
- To make the text intelligible again, reverse the arguments
  - tr ‘[N-Z][A-M][n-z][a-m]’ ‘[A-M][N-Z][a-m][n-z]’
- The -d option will cause tr to delete selected characters
  - echo If you can read this you can spot the missing vowels | tr -d ‘aeiou’

tr Gotchas

- The syntax varies between BSD and System V
  - BSD uses a-z
  - System V uses ‘[a-z]’
  - System V allows [*x***n*] to indicate *n* occurrences of *x*
    If you leave out *n*, then *x* is duplicated as many times as necessary
  - In BSD, if *string2* does not contain as many characters as *string1*, the last character of *string2* is duplicated as many times as necessary
  - System V fails to translate these additional characters
- Would *tr* solve the problem of capitalizing all first letters of each word in a sentence?

What If?

- Suppose you need a tool that will, tidy up corrupt or badly formed web scrapings:
  - Accept any number of files to check, report each line that has doubled words, highlight the doubled words (using ANSI escape sequences) and display the file name with each line
  - Work across lines even when the word at the end of a line is repeated at the beginning of the next line
  - Find doubled words despite capitalization, punctuation, or amount of separating whitespace
  - Find doubled words even if they are separated by HTML tags
- Unlikely to be so bad, but as an example...

Possible in a few lines, if you know...

- The following is merely a perverse example of power play from nerds…and not to be studied, unless you are that way inclined.
- Using the regular expression engine from Perl, this problem can be solved in only a few lines of code
- And is possible with other implementations of *regular expressions*

```
$/ = ".\n" ;
while (<>) {
  next if !s/\b([a-z]+)((\s|<[^>]+>)+)(\1\b)/\e[7m$1\e[m$2\e[7m$4\e[m/ig;
  s/^([^\e]*\n)+//mg;
  s/^/$ARGV: /mg;
  print;
}
```

Code was written by Jeffrey E. F. Friedl as an example in ‘*Mastering Regular Expressions*’ O'Reilly Publishing

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
- For example, to find our doubled words example
  - \<([A-Za-z]+)sp+ \1\>'
  - This first finds a generic word \<([A-Za-z]+)' followed by one or more spaces sp+'
  - The \1\>' is then interpreted & evaluated as follows:-

    \1 - denotes the first subexpression just defined \<([A-Za-z]+)'

    \> … as the end of a word

    Which effectively means that consecutive duplicated words are matched

Coding code…

- Code
  - for clarity & comprehension,
  - not for concise, complexity & confusion
- Powerful expressive often cryptic languages,
  - say much with little,
  - tend to be notoriously perverse, error prone, practically impenetrable,
  - are virtually impossible to maintain, unless commented
- At the opposite extreme, less expressive languages need more code, with increasing likelihood of error also.

Regular Expressions

- You can use and even administer Unix systems without understanding *regular expressions* but you will be doing things the hard way
- *Regular expressions* are endemic to Unix
  - vi, ed, sed, and emacs
  - Awk, Tcl, Perl and Python, SQL … even search engines
  - grep, egrep, fgrep
- *Regular expressions* descend from a fundamental concept in Computer Science called *regular grammars*, from *finite automata* theory
  - Just a fancy way of defining & validating a sequence of tokens

So What Is a Regular Expression?

- A *regular expression* is simply a description of a pattern that describes a set of possible characters in an input string
- Have already seen some simple examples of *regular expressions* (known as *regex* from here on)
  - In vi when searching :/c[aou]t searches for cat, cot, or cut
  - In the shell

    ls *.txt

    » but in string regexp c* implies 0 or more c's

    cat chapter?

    cp Week[1234].pdf /home/monthly

Downside of Regular Expressions

- There is considerable variation from utility to utility
  - The shell is limited to fairly simple metacharacter substitution (*,?, [···]) and doesn' t really support regex
  - Regex in *ed* and *vi* are also fairly limited
  - Regex in *sed* are not exactly the same as regex in *Perl*, or *Awk*, or *grep*, or *egrep*
- This puts the onus on the user to examine the man page or other documentation for these utilities to determine which flavor of regex are supported

Giving rise to the quip...! >

If you have a problem;

- and the solution requires regular expressions,

Now you have another problem!

So How Do We Build a Regex?

- The simplest regex is a normal character
  - c , for example, will match a c anywhere while an a will do the same for an 'a'
- The next thing is a . (period)
  - This will match any single occurrence of any character except a newline
  - For example . will match a 'z' or an 'e' or a '?' or even another '.'
  - w.n will match 'win' , wan' , 'won' , 'wen' , 'wmn' , 'went' , and 'wanton' as well as 'w*n' and 'w9n'
- Complex regex are constructed by simply by stringing together smaller simpler regexs

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, sometimes we still want to use a special character as itself
- To do this, we "escape" the character with a \ (backslash)
- Suppose we want to search for the character sequence '8*9*'
  - Unless we do something special, this will match zero or more '8's followed by zero or more '9's, not what we want
  - '8*9*' will fix this - now the asterisks are treated as regular characters

Multiple Occurrences in a Pattern

- The * (asterisk or star) is used to define zero or more occurrences of the single character preceding it
  - 'abc*d' will match 'abd' , 'abcd' , 'abccd' , 'abcccd' , or even 'abccccccccccccccccccccccccccccccccccd'
  - Note the difference between the * in a regex and the shell's usage
    - In a regex, a * only stands for zero or more occurrences of a single preceding character,
    - In the shell, the * stands for any number of characters that may or may not be different

Specifying Begin or End of Line

- The '^' specifies the beginning of a line
  - '^The' then will match any 'The' that are the first characters on a line
- The '$' matches the end of line
  - 'well$' will match 'well' only if they are the last characters on a line prior to the NEWLINE character
  - Note that 'well ' (notice the space at the end) would NOT match 'well$'
- '^Bin$' would only match a line that started with 'Bin' and then had no other characters on the line
- What would the regex '^$' do?

Character Classes [ ]

- The square brackets [ ] are used to define character classes
  - '[aeiou]' will match any of the characters 'a' , 'e' , 'I' , 'o' , or 'u'
  - '[aA]wk' will match 'awk' or 'Awk'
- 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', '3'

Negating a Character Class

- The '**^**', when used as the first character in a character class definition, negates the definition
  - For example '[^aeiou]' matches any character except 'a' , 'e' , 'I' , 'o' , or 'u'
  - Used anywhere else within a character class, the '^' simply stands for itself '^'
  - '[ab^&]' matches a 'a' , 'b' , '^' , or '&'
  - Note also that within a character class, the '^' does not stand for beginning of line

Reading a Regex

- If you get in the habit of literally reading a regex, it will be much easier for you to determine what one does
  - '^Line' could be read as matching "the word Line at the beginning of a line'
  - A better way to read it is "the beginning of a line followed by a capital L followed by an i ... n … e"
    - actually how the finite state automata interprets its
  - '^corn$' would be read as "the beginning of a line followed immediately by a c followed by an o followed by an r followed by an n followed immediately by a NEWLINE"
-

Alternation

- Regex also provides an alternation character (|) for matching one or another subexpression
  - '(T|P)en' will match 'Ten' or 'Pen'
  - '^(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 ':'

    By the way, mail is normally stored as sequential text file, with new messages beginning with new lines From etc.

    Mail messages can be separated from underlying text with regex.
- The parenthesis ( ) are used to limit the scope of the alternation
  - 'At(ten|nine)tion' then matches "Attention" or "Atnination", not "Atten" or "nination" as would happen without the parenthesis - 'Atten|nination'

Word Boundaries

- The regex 'cat' will match **cat**, con**cat**enate, **cat**astrophe, and **cat**atonic
- What if I only want to match the word "cat" ?
- Some regex flavors implement the concept of words
  - '\< ' signifies the beginning of a word and '\>' signifies the end of a word
  - These aren' t metacharacters but when used together the have special meaning to the regex engine

- Note that the regex engine does not understand English
  - A beginning-of word is just the position where a sequence of alpha numeric characters begin
  - End-of-word is where the sequence stops
- Where is that &^%$# stinkin' roadrunner-lovin' coyote?

Optional Items

- The '?' (question mark) specifies an optional character, the single character that immediately precedes it
  - For example, if I am looking for the month of July, it may be specified a "July" or "Jul"
  - I could use '(July|Jul)' to search or I could use 'July?'

Repetition

- The * (asterisk or 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’ while ‘abc?d’ will match ‘abd’ and ‘abcd’ but not ‘abccd’
- 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

Remember – memory aids...

‘*’

(asterisk or star) is often associated with multiplication, and you can multiply by zero to get zero!

‘+’

is associated with (plus),
So you can add to it, not easy to take away or to zero.

‘?’

Is associated with doubt,
Might or might not be exist….

Repetition Ranges

- Ranges can also be specified
- {n,m} 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

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
- For example, to find our doubled words example
  - \<([A-Za-z]+)sp+ \1\>’
  - This first finds a generic word \<([A-Za-z]+)’followed by one or more spaces sp+’
  - The \1\>’is then interpreted & evaluated as follows:-

    \1 - denotes the first subexpression just defined \<([A-Za-z]+)’

    \> … as the end of a word

    Which effectively means that consecutive duplicated words are matched

Regex Summary

Character	Name	Meaning
.	dot	any one character
[?]	character class	any character listed
[^?]	negated character class	any character not listed
^	caret	position at start of line
$	dollar	position at end of line
\<	backslash less-than	position at beginning of word
\>	backslash greater-than	position at end of word
?	question mark	matches optional preceding character
*	asterisk or star	matches zero or more occurrences
+	plus sign	matches one or more occurrences
{n,m}	n to m	matches m to n occurrences
|	bar, or	matches either expression it separates
(?)	parenthesis	limits scope of | or encloses subexpressions for backreferencing
\1, \2, ?	backreference	Matches text previously matched within first, second, etc set of parenthesis

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)

  Note that hours are alternated to double or single digit

grep

- *Grep* helps find files which contain specific text
- Most GUI's provide a 'find' interface for grep.
- grep comes from the ed 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 other variants of grep, deprecated but retained for legacy code
  - *egrep* extended grep – same as : grep -E
  - *fgrep* fixed grep – same as : grep -f
  - *Rgrep* recursive grep – same as : grep -r

43

grep Family Differences

- Grep – uses Basic Regular Expressions (BRE)
  - uses regular expressions for pattern matching
- fgrep - file grep – give a list of words, forget re's!
  - does not use regular expressions,
  - only matches fixed strings
  - but can get search strings from a file
- egrep – exponential/extended grep,
  - uses a more powerful set of regular expressions (Extended rather than Basic RE)
  - but does not support backreferencing,
  - generally the fastest member of the grep family
    - Better algorithms and no backreference tracking

44

grep Family Syntax

grep [-hilnw] [-e expression] [filename]
egrep [-hiln] [-e expression] [-f filename] [expression] [filename]
fgrep [-hilnx] [-e string] [-f filename] [string] [filename]

- -h - Do not display filenames
- -i - Ignore case – saves all [:upper:] [:lower:] [Aa] or [a-zA-Z]
- -l - List only filenames containing matching lines
- -n - Precede each matching line with its line number
- -w - Search for the expression as a word (*grep* only)
- -x - Match whole line only (*fgrep* only)
- -e expression - Same as a plain expression but useful when expression starts with a –
- -e string - fgrep only uses search strings, no regular expressions
- -f filename - take the regular expression (egrep) or a list of strings separated by NEWLINES (fgrep) from *filename*

45

Regex in the grep Family

- The following one-character regexs match a single character
  - c - an ordinary character
  - \c - an escaped special character . * [\ ^ $
  - A \ followed by < > ( ) { or }
  - A . (period)
  - [string] any single character contained within the brackets

46

Rules For Constructing grep Regex

- A single character regex followed by a '*' matches zero or more occurrences of the single-character regex
- A regex enclosed in \(and \) matches whatever the regex matches and tags it (grep only)
- \n matches the same string the corresponding n^{th} previous \(regex\) matched
- The concatenation of regexs is a regex that matches the concatenation of the strings matched by each component of the regex
- A regex followed by a \{m\}, \{m,\}, or \{m,n\} matches a range of occurrences of the regex

47

egrep Regex

- *egrep* uses the same rules except for \(, \), \n, \<, \>, \{, and \}
- *egrep* adds the following regex components
  - '*' a regular expression followed by a '*' matches zero or more occurrences of the expression, not just a single character
  - '+' a regex followed by a '+' matches one or more occurrences of the regex
  - '?' a regex followed by a '?' matches zero or one occurrences of the regex
  - '|' provides alternation: two regex separated by | match either a match for the first or the second regex
  - a regex enclosed in () provides a match for the regex

48

Regexs for grep and egrep

RegExp	Meaning
c	Normal (nonmeta) character
\m	Escape character
^	Start of line
$	End of line
.	Any single character except newline
[x y z ?]	Any of x, y, z, ?
[a-z]	Range: a..z
[^?]	Any single character not listed

Regexs for grep and egrep

RegExp	Meaning
\n	The n[th] tagged expression.
r*	Zero or more r's (* implies 0 or more… * by 1 no change: omit)
r+	One or more r's (egrep only) (+ implies at least one, + 0 no change:omit)
r?	Zero or one r's (egrep only) (? Implies either there or not; so (1,0))
r1 r2	concatenation : r1 followed by r2
r1| r2	r1 or r2
\(r\)	Tagged regular expressions r; (grep only)
(r)	Regular expression r

grep Family options

Option	Action
grep, fgrep and egrep	
-h	suppress filename display
-l	only display filenames with matches found
-n	number each matching line
-i	ignore case
-e	match expression, useful when expression begins with a -
grep only	
-w	match whole words only
egrep and fgrep	
-f filename	match strings stored in filename
fgrep only	
-x	match whole line exactly

grep Family Expressions

RegExp	**Matches**
grep, fgrep and egrep	
x	Ordinary characters match themselves, - except Newlines & metacharacters
xyz	ordinary strings match themselves.
grep and egrep	
\m	match literal character 'm'
^	start of line
$	end of line
.	Any single character
[xy^$z]	any listed…x,y,^,$,z
[^xy^$z]	any character except x,y,^,$,z
[a-z]	any character in the given range
[^a-z]	any character **not** in the given range

More grep Family Expressions

RegExp	Matches
grep only	
\(r)\	Tagged regular expression matches r
\n	Set to what matched the nth tagged expression (0≤n≤9)
egrep only	
r+	one or more occurrences of r
r?	Zero or more occurrences of r
r1|r2	either r1 or r2
(r1|r2)r3	(Either r1 or r2) followed by r3 …r1r2 or r1r3
(r1|r2)*	Zero or more occurrences of (r1 or r2) e.g. r1, r1r1, r2r1basically as many rx as you want with x:(1,2)

grep Examples

- grep 'the' filename
- grep -w 'the' filename
- grep '\<the\>' filename
- grep 'fo*' filename
- egrep 'fo+' filename
- egrep -n '[Tt]he' filename
- grep -nw '[Tt]he' filename
- fgrep 'The' filename
- egrep 'NC+[0-9]*A?' filename
- fgrep -f expfile filename
- fgrep -x 'The End' filename