Note grep default

Default is to output

each line with a match

only once

Regardless of

how many matches occur in the line

There are GUI alternatives such as
- PowerGrep limited to windows
- and cross platform ones such as DocFetcher & Fsearch
but stick to syllabus, scriptable and general use grep..

Grep

- Used in textual analysis – even by linguists!
  - Information retrieval … stemming..
  - Verify authorship via style based on word frequencies & etc..
- Faster than having to (re-)code in C, Java…for most problems – depends on problem and skill.
- Runs very fast
  - Honed fast algorithms – arguably still the best
  - Underlying algorithms may change/vary as needed
  - Optimised over the years in Unix, even to the point of streaming the file through buffers & cache
  - several studies support this (perhaps biased !?)
    - Find the refn. Or suggest a search
  - Jibe: Google Search is just a massively global grep!?
    - G used human intelligence of pages followed to rank relevance! (Now AI)
- 'biogrep' 2009 – multithreaded POSIX threads, GNU regex, fast genome search
  - Human genome ~ 200GB; expect 4 x 10^10 GB for genomic data ~2025

grep

- grep comes from the ed search command
  global **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 variants of grep (POSIX !? ;-)
  - *grep* basic regex or almost extended with grep -E
  - *egrep* extended grep – with extended regex
  - *fgrep* fixed grep – just takes strings
- *grep* is the answer to the moments where you know you want a the file that contains a specific phrase but you can't remember it's name (e.g. OS X Spotlight search & other OS equivalents)
- Most OS GUI's provide a 'find' interface for grep.
- These [ef]grep variants are being deprecated in favour of flags:-
  : grep -e for egrep, grep -f for fgrep etc

grep Family Differences

- grep –
  - uses regular expressions for pattern matching
- grep –f (formerly fgrep - fixed/file grep)
  - does not use regular expressions,
  - only matches fixed strings
  - ***can get a list of search strings from a file***
- grep –E (formerly egrep - extended)
  - uses extended regular expressions
    - Easier syntax, more flexible & powerful

Backreferences & obscure regex may require exponential time

- Needs to maintain dynamic trace of matches
  - Requires exponential time and space
  - As any new regex term match
    - Must be compared with all previous
    - So the $(n+1)^{th}$ match searches all past n matches
      - Which happens for each new match
        - Sum of integers from 1 to n
          - => ~ $n^2/2$ (exponential (power of 2) in n...)
          - Both space and time for simple brute force search
          - Although faster searches & indexing possible

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 – (hide) - do not display filenames
- -i - Ignore case – very handy saves all this: [Aa] or [a-zA-Z]
  (check as 'i' often means interactive with many commands!)
- -l - List only filenames containing matching lines for popular terms
- -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 '–', or put '-' first in list of [-chars]text
- -e string - fgrep only uses search strings, no regular expressions
- -f filename - take the search criteria from a file 'filename'
  - Either a list of regular expression (egrep)
  - or a list of strings separated by NEWLINES (fgrep)

Regex in the grep Family

- The following one-character regexs match a single character
  - c - an ordinary character
  - \c - an escaped special character . * [\ ^ $
    - (when you want to literally match one of the chars '.', and not their special control char meaning
      - E.g. '.' – as regex meta means any single char but as \. just means a full-stop/period
  - \ followed by brackets
    - \< \> start or end of word
    - \( )\ tags for backreference
    - { or } specifies subsequent (re|oc)currence of \( )\
  - [string] any single character contained within the brackets

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 (n > 0) (grep only) - for backreference
- \n matches the same string the corresponding n^{th} \(regex\) matched – the actual backreference... reference!
- The concatenation of regexs is a regex that matches the concatenation of the strings matched by each component regex
- A regex followed by a \{m\}, \{m,\}, or \{m,n\} matches a range of occurrences of the regex
- Script arguments, $n, can be passed to regular expressions within single quotes e.g. ['$n'

3 – ‘modern’ extended Regex : egrep – grep -e

- *GNU grep gives basic & extended regex, same functionality, does not apply generally e.g. BSD (OS X)*
- *egrep* (extended regex) uses the same rules except for :
  if you want a bracket, just use a bracket - no backslash
  just use ( ) < > { } not \(, \), \<, \>, \{, and \}
- *egrep* adds the following regex components
  - a regular expression
    followed by any of the following character multipliers *, +, or ? matches the same respective multiples of the expression,
    (*, +, ?) (>=0, >0, 0 or 1) not just a single character
  - | 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

Trust manuals & implementations !?

Further comments on AND & OR with grep regex in relevant lab notes.
The GNU manual implies grep supports lookaround (necessary for regex AND)

- With Perl compatible regex, using a -P flag
- But does not (always) implement it (depending on configuration/compilation),
- but may be iffy as some patches are not accepted yet
  - Especially with respect to logical negation
- And this support has been removed in MacOSX from 10.8 apparently
  - But then that’s Unix BSD
- So probably best ignored, and didn’t bother with it

But checked it out...to see and, it worked !!!

- all Ubuntu Long Term Support (LTS) >= (16-20).04 seem to support the basic

```
$ echo snowflake | grep -P '(?=.*lake)(?=.*snow)'
snowflake
```

snowflake : first string thought of with 2 simple words & internal separator

Fairly Rare regex – some show off will use!

(?#...) Comment, "..." is ignored.

(?:...) Matches but doesn't return "..."

Lookahead

- general area also referred to as ‘lookaround’

- any correlation between information and implementation is probably accidental – manual and commands may differ!

(?=...) Matches if expression would match "..." next

(?!...) Matches if expression wouldn't match "..." next

Regexs for grep and egrep – semi-formal manual style

regex	**Meaning**
c	Normal (nonmeta) character
\m	Escape a metacharacter to take it literally as ‘m’
[a-z]	Range: a..z
^	Start of line within regex even within ( ) but not in [ ] = NOT if first within a square b[^list] or range [^a-e]
[^?]	Complement of list single character not listed (? Represents the character list in this case!) - take care when reading notes as notation can vary
$	End of line
.	Any single character except newline
[x y z ?]	Any of x, y, z, ?
c?	char may or may not be there 0 or 1 occurrence

Regexs for grep and egrep – semi-formal manual style

regex	Meaning
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	alternation r1 or r2
(r)	Regular expression r
\(r\)	Tagged regular expressions r; (grep only)
\n	The n^{th} tagged expression.

grep Family options

Option	*Action*
grep, fgrep and egrep	
-h	suppress filename display
-l	only display filenames with matches found
-i	ignore case
-e	match expression, useful when expression begins with a -
-n	number each matching line
-o	output each exact match found in a line on a separate line... - useful if you need to count matches, and not just find lines
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 - examples

regex	*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 - the caret ^ is in the middle, not the first
[^xy^$z]	any character except x,y,^,$,z – as the caret ^ is first negating the rest
[a-z]	any character in the given range
[^a-z]	any character **not** in the given range

'^[abc]$'
matches a line with only one of a,b,c

'^[^abc]$'
matches a line with any one character except for a, b, c.

More grep Family Expressions

regex	*Matches*
grep only	
\(r)\	Tagged regular expression matches r, for backreferences
\n	Set to what matched the n^{th} tagged expression ($0 \le n \le 9$)
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, r2r1 basically as many rx as you want with x:(1,2)

grep Examples on a file: GrepText

For sample text, just redirect the manual on 'grep' to GrepText as a textfile to tes grep commands below, by issuing the command: (repeat with 'mat', not 'text')

man grep >GrepText

grep	'text '	GrepText	find text – will include context
grep	–w 'text '	GrepText	find the word text
grep	'\<text \>'	GrepText	again find the word 'text'
grep	'fo*'	GrepText	any of : f, fo, foo,
egrep	'fo+'	GrepText	any of : fo, foo, etc., but not just 'f'
egrep	-n '[Tt]ext'	GrepText	show lines + numbers with results'
grep	-nw '[Tt]ext'	GrepText	as previous but results are words
fgrep	'text'	GrepText	fgrep : fixed strings NOT regexs
egrep	'NC+[0-9]*A?'	GrepText	N, at least one C, any no. of digits, followed by an optional 'A' {0 or 1}
fgrep -f expfile		GrepText	search for strings from lines in file
fgrep -x 'for text'		GrepText	find lines with only 'for text'

regex – catch summary

- * in shell is a wildcard : any no. of any chars : anything
- * in regex is a quantifier : any no. (≥0) of previous character
- ^ - means start of line, unless
- Within [ ] where
  - If at start it negates (i.e. anything but) the character set
  - Otherwise just means itself within the set of characters
- Chaos : Different flavours exist for
  - Languages : Java, Javascript, .NET, PHP
  - Variants of greps
- Some order
  - web, apps etc. …for language specific regex testers
  - Or mess with ed, vim etc..

Language use of regex

- Most programming languages support their own version of regex…but need coding & compilation
  - To aid text processing
  - Parse text : records, fields, dates, serial nos. etc..
  - Scraping information from websites,
  - Newsfeed processing
- Access and standards vary widely,
  - best to consult the appropriate language support helpfiles or tutorial
  - Most programming languages support calls using regex to match, search, replace, split strings etc.,

Working regex to validate email addresses..

1. Dirt-simple approach

Here's a regex that only requires a very basic *xxxx@yyyy.zzz*:

.+\@.+\..+

in English: (>1 of any char)@(>1 of any char)(literal .) (>1 of any char)

Upside: Dirt simple.

Downside: Accepts invalid addresses: xxxx@yyyy.zzz, or even a@b.c,

2. Slightly more strict (but still simple) approach

[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}	range limits 2-4chars
\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b	\b word boundary
^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$	^ $ end of line anchors

Upside: Only allows email address-friendly characters, restricts domain extension to only two to four characters.

Downside: It still allows many invalid email addresses, and misses some longer domain extensions (e.g. .museum)

Working regex to validate email addresses..

3. Specify all the domain extensions approach

```
([a-z0-9][-a-z0-9_\+\.]*[a-z0-9])@([a-z0-9][-a-z0-9\.]*[a-z0-9]\.(arpa|root|aero|
biz|cat|com|coop|edu|gov|info|int|jobs|mil|mobi|museum|name|net|org|pro|
tel|travel|ac|ad|ae|af|ag|ai|al|am|an|ao|aq|ar|as|at|au|aw|ax|az|ba|bb|bd|be|
bf|bg|bh|bi|bj|bm|bn|bo|br|bs|bt|bv|bw|by|bz|ca|cc|cd|cf|cg|ch|ci|ck|cl|cm|cn|
co|cr|cu|cv|cx|cy|cz|de|dj|dk|dm|do|dz|ec|ee|eg|er|es|et|eu|fi|fj|fk|fm|fo|fr|ga|
gb|gd|ge|gf|gg|gh|gi|gl|gm|gn|gp|gq|gr|gs|gt|gu|gw|gy|hk|hm|hn|hr|ht|hu|id|ie|
il|im|in|io|iq|ir|is|it|je|jm|jo|jp|ke|kg|kh|ki|km|kn|kr|kw|ky|kz|la|lb|lc|li|lk|lr|ls|lt|lu|
lv|ly|ma|mc|md|mg|mh|mk|ml|mm|mn|mo|mp|mq|mr|ms|mt|mu|mv|mw|mx|
my|mz|na|nc|ne|nf|ng|ni|nl|no|np|nr|nu|nz|om|pa|pe|pf|pg|ph|pk|pl|pm|pn|pr|
ps|pt|pw|py|qa|re|ro|ru|rw|sa|sb|sc|sd|se|sg|sh|si|sj|sk|sl|sm|sn|so|sr|st|su|sv|
sy|sz|tc|td|tf|tg|th|tj|tk|tl|tm|tn|to|tp|tr|tt|tv|tw|tz|ua|ug|uk|um|us|uy|uz|va|vc|ve|
vg|vi|vn|vu|wf|ws|ye|yt|yu|za|zm|zw)|([0-9]{1,3}\.{3}[0-9]{1,3}))
```

Upside: It doesn't allow xxxx@yyyy.zzz!

Downside: Needs continuous updating! (or use a file & fixed pattern grep)

Official email standard RFC 5322 incomplete

No check of top-level domain names, too broad, unsupported.

NB IP addresses validated within square brackets below….

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*
  | "(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]
  | \\[\x01-\x09\x0b\x0c\x0e-\x7f])*")
@
(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?
  | \[
        (?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
        (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?|[a-z0-9-]*[a-z0-9]:
        (?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]
        | \\[\x01-\x09\x0b\x0c\x0e-\x7f])+)
    \])
```

Working regex to validate email addresses..

4. But there are still problems

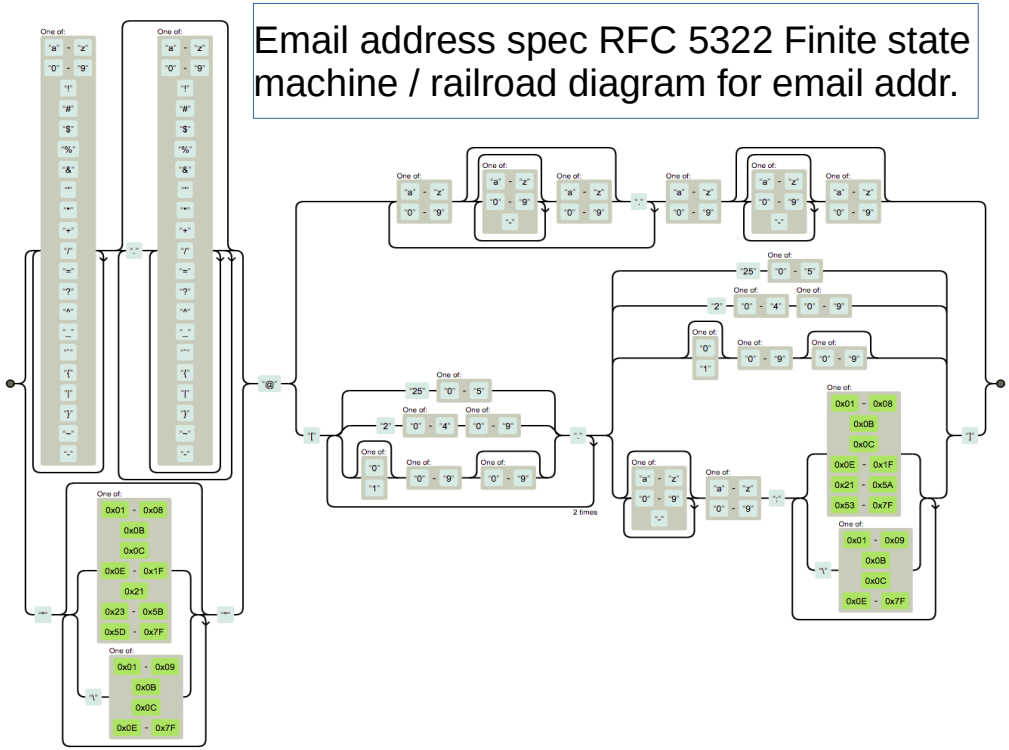
(?#...)	Comment, "..." is ignored.
(?:...)	Matches but doesn't return "..."
(?=...)	Matches if expression would match "..." next
(?!...)	Matches if expression wouldn't match "..." next

Although the foregoing will match the likes of
john@server.department.company.com
It will also accept the likes of the following, which is invalid:
john@aol...com
(Presentation software (ppt & odp) colours both as valid emails – ! ~ 2015)

This can be corrected by replacing [A-Z0-9.-]+\.
With (?:[A-Z0-9-]+\.)+
to validate strings after '@', which should only have one '.' to give this :
\b[A-Z0-9._%+-]+@(?:[A-Z0-9-]+\.)[A-Z]{2,4}\b (error had + in past at !)

Tradeoff between & cost & requirements dictate an acceptable solution!
False positives : accepting what is unacceptable
False negatives : not accepting what is acceptable

Email address spec RFC 5322 Finite state machine / railroad diagram for email addr.



Working regex to validate email addresses..

Easier and more pragmatic to ignore the double quotes and square brackets of the official specification which will validate almost all

[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*
@
(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?

And can be improved by specifying top-level domain names, demonstrated a few slides ago.

So, the answer is : there is no answer; except what works for you!

Check it out on your own data and situation,

Or it may check you out!

Why bother .. So don't!?

- Validating emails is a pointless exercise!
- Don't waste your time!
- Why would you want to…
  - So you check emails for subscriptions
  - To avoid sending spam in your marketing
- When there are other ways?
  - Get them to subscribe by entering their email
  - Have them validate it themselves… double-entry
- What's the worst that can happen…
  - Spam…!?

Regex in PowerShell (Windows shell)

The following command finds all files with

- "PowerShell" or "Hyper-V" in the name
- extensions .docx or .txt

Note that, in PowerShell pipes, regex and output are similar to 'nixes, flags are longer with meaning, but verbose flags are also in Linux the '--flags'

Since PowerShell is object-oriented, appropriate object methods are required to match expressions, rather than universal text for Linux.

```
PS D:\> Get-ChildItem –Recurse -File |
  Where-Object {(($_.Name -match '.*PowerShell.*\.(docx|DOCX)\b')
     -or ($_.Name -match '.*Hyper-V.*\.(doc|DOC|docx|DOCX)\b')) }
```

```
Directory: D:\
21-6-2012    00:52  Practical PowerShell.docx
12-7-2012    18:32  PowerShell ft Hyper-V.docx
16-7-2012    12:16  Hyper-V Beyond.doc
```

Variations on dir will do the same as Get-ChildItem, displaying children within subdirectory nodes etc.

AWK's extended regex manual extract – specifying characters

AWK uses extended regular expressions like egrep,
with these metacharacters : ^ $. [ ] | ( ) * + ?

Regular expressions are built up from characters as follows:

c	matches any non-metacharacter c.
\c	matches a character defined by the same escape sequences used in string constants or the literal character c if \c is not an escape sequence.
.	matches any character (including newline).
^	matches the front of a string.
$	matches the back of a string.

[c1c2c3...] matches any character in the class c1c2c3... .
An interval of characters is denoted c1-c2 inside a class [...].
[^c1c2c3...] matches any character not in the class c1c2c3...

Compound regex – extended (&GNU BRE)

Regular expressions are built up …

… from other regular expressions as follows:

r1r2 matches r1 followed immediately by r2 (concatenation).

r1 | r2 matches r1 or r2 (alternation).

r* matches r repeated zero or more times.

r+ matches r repeated one or more times.

r? matches r zero or once.

(r) matches r, providing grouping.

The increasing precedence of operators is:-
alternation, concatenation and unary (*, + or ?).

Examples

AWK identifiers /^[_a-zA-Z][_a-zA-Z0-9]*$/ and

AWK numbers /^[-+]?([0-9]+\.?|\.[0-9])[0-9]*([eE][-+]?[0-9]+)?$/

Note that . has to be escaped to be recognized as a decimal point, and that metacharacters are not special inside character classes. [ ]

Any expression can be used on the right hand side of the ~ or !~ operators or passed to a built-in that expects a regular expression. If needed, it is converted to string, and then interpreted as a regular expression. For example,

```
BEGIN { identifier = "[_a-zA-Z][_a-zA-Z0-9]*" }
$0 ~ "^" identifier
```

prints all lines that start with an AWK identifier.

mawk matches the empty regular expression, //,

I.e the empty string at the front, back and between every character: e.g., using echo, pipe & global substitution:-

```
echo abc | mawk '{ gsub(//, "X") ; print }'
XaXbXcX
```

Finally

Of course, always check, inspect and test,
before recklessly wrecking your best!
There are online, and downloadable regex testers!

- Regexpal, regexpbuddy
- Regexmagic

And possibly the largest range of apps for regex

- http://www.regexplanet.com/

And a great resource

- http://www.regular-expressions.info/
- And even an Android app for regex!

But best of all test on your own or the intended system
i.e. LOOK BEFORE YOU LEAP!
YOU ARE NOT EXPECTED TO BE FAMILIAR WITH ALL OF THIS
BUT JUST BE AWARE OF IT'S EXISTENCE SHOULD YOU EVER NEED IT!

Auto-generated 6,000 chars in Perl … for 4 byte IP regex


```
$RFC822PAT = <<'EOF';
[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*(?:[^(\040)<>@,;:".\\[]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\[]\000-\037\x80-\xff])|"[^\\\x80-\xff\n\015"]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015"]*)*"[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*(?:\.[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*(?:[^(\040)<>@,;:".\\[]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\[]\000-\037\x80-\xff])|"[^\\\x80-\xff\n\015"]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015"]*)*"[\040\t]*)*)*@[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*(?:[^(\040)<>@,;:".\\[]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\[]\000-\037\x80-\xff])|\[(?:[^\\\x80-\xff\n\015\[\]]|\\[^\x80-\xff])*\][\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*)(?:\.[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*(?:[^(\040)<>@,;:".\\[]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\[]\000-\037\x80-\xff])|\[(?:[^\\\x80-\xff\n\015\[\]]|\\[^\x80-\xff])*\][\040\t]*)*)*<[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*(?:[^(\040)<>@,;:".\\[]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\[]\000-\037\x80-\xff])|"[^\\\x80-\xff\n\015"]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015"]*)*"[\040\t]*)*(?:@[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*(?:[^(\040)<>@,;:".\\[]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\[]\000-\037\x80-\xff])|\[(?:[^\\\x80-\xff\n\015\[\]]|\\[^\x80-\xff])*\][\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*)(?:\.[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*(?:[^(\040)<>@,;:".\\[]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\[]\000-\037\x80-\xff])|\[(?:[^\\\x80-\xff\n\015\[\]]|\\[^\x80-\xff])*\][\040\t]*)*)*:[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*)?(?:[^(\040)<>@,;:".\\[]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\[]\000-\037\x80-\xff])|"[^\\\x80-\xff\n\015"]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015"]*)*"[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*)*(?:\.[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*(?:[^(\040)<>@,;:".\\[]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\[]\000-\037\x80-\xff])|"[^\\\x80-\xff\n\015"]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015"]*)*"[\040\t]*)*)*:[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*(?:[^(\040)<>@,;:".\\[]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\[]\000-\037\x80-\xff])|\[(?:[^\\\x80-\xff\n\015\[\]]|\\[^\x80-\xff])*\][\040\t]*)*)?;[\040\t]*)*
```


8 hex digits + 3 colons = 11 chars needs >6,000 autogenerated chars in Perl as verification… had dumped this slide… but just in case you thought ICT is smart!?