Pattern Matching

Using Regular Expressions

Nick Maclaren, Computing Service

Most of this is the work of Philip Hazel

Beyond the course

- The speaker:

  Nick Maclaren, nmm1@cam.ac.uk, ext. 34761

- The foils, some examples etc.:

  http://www-uxsup.csx.cam.ac.uk/courses/REs

- Best Email to use for advice:

  escience-support@ucs

- A book on the current practice

  Mastering Regular Expressions, Third Edition
  Jeffrey E.F. Friedl, O'Reilly

- See also the theory section at the end of the handout
  Specifically the Wikipedia reference on BNF

Practice makes perfect

- To really learn regular expressions you need practice
- An experimental “exerciser” is available at

  `http://www-uxsup.csx.cam.ac.uk/...`
  `.../courses/REs/phreex`

- This is a Perl script that does line-by-line interaction
  - It is not a web-based application
  - You need to download it in order to run it
- You are asked to write expressions which are then tested

What is a regular expression?

A regular expression is a *Pattern* or *Template*

for matching a set of text strings

Humans are good at recognizing shape patterns

Which includes reading meaningful text

Even when thoroughly obfuscated!

But it’s not always obvious...

Aside: a fascinating book

- If you want to learn more about this, read

  Francis Crick
  The Astonishing Hypothesis: the scientific search for the soul

- The title is complete and utter codswallop
  - The blurb blithers on about consciousness
- It is entirely about visual perception
  - Based on studying the visual cortex
- It explains the effect in the previous slide
  - You can *scan* on colour or shape, but not both
  - You have to *search* for a combination

Matching text is hard for humans

- Humans are not good at matching random text

  `CAGTACGGGTCACTAGAAAATGAGTATCCTCGAATTGCTATCCG`

- Can you spot `ACGT` above?
  (In fact, it is not present)
- Can you spot the the typo in this sentence?
  (It's easier than some)

- Computers can do a much better job at matching text
  If you give them the right instructions...
- Regular expressions are powerful matching instructions
  They are really little computer programs
  There is scope for writing good ones and bad ones

Patterns that are not regular expressions

- Some common computing patterns are not regular expressions

```
/usr/share/*
myphotos/pict00??.jpg
```

- Those are examples of filename *globs* or *wildcards*
- Regular expressions are much more generalized patterns
- Globs cannot help with requirements such as

  Match “grey” or “gray”
  Match “foetus” or “fetus”
  Match dates like “12-Feb-07” or “12-02-07”
  Match HTML tags like “<p>” or “ ”

Origin of Regular Expressions

- Regular expressions come from mathematics
  - They arose in finite state automata theory in the 1950s
- Used for defining programming languages and data formats
  - We will return to this aspect at the very end
- They migrated into early text editors such as *qed* and *ed*
  - Unix tools such as *grep* and *awk* also used them
- Various flavours started to evolve

- The “Cambrian Explosion” of regex features came with Perl
  - Especially Perl 5 in 1994
- No longer “regular expressions” in the mathematical sense
  - The term has been hijacked in common usage

Regular Expression Flavours

- Regular expressions are used in many programs
  - Various versions of *grep*
  - Various text editors
  - Perl, Python, Java, and other languages
- There are variations in syntax and semantics
- PCRE (Perl-Compatible Regular Expressions) is a library
  - Used by Apache, Exim, Mathematica, nmap, PHP, tin, ...
- This course concentrates on Perl/PCRE regular expressions
  - Other flavours may use different syntax
  - The underlying principles are what matters
- These web sites have some discussion and examples
  - http://www.regular-expressions.info/examples.html
  - http://regexlib.com/default.aspx

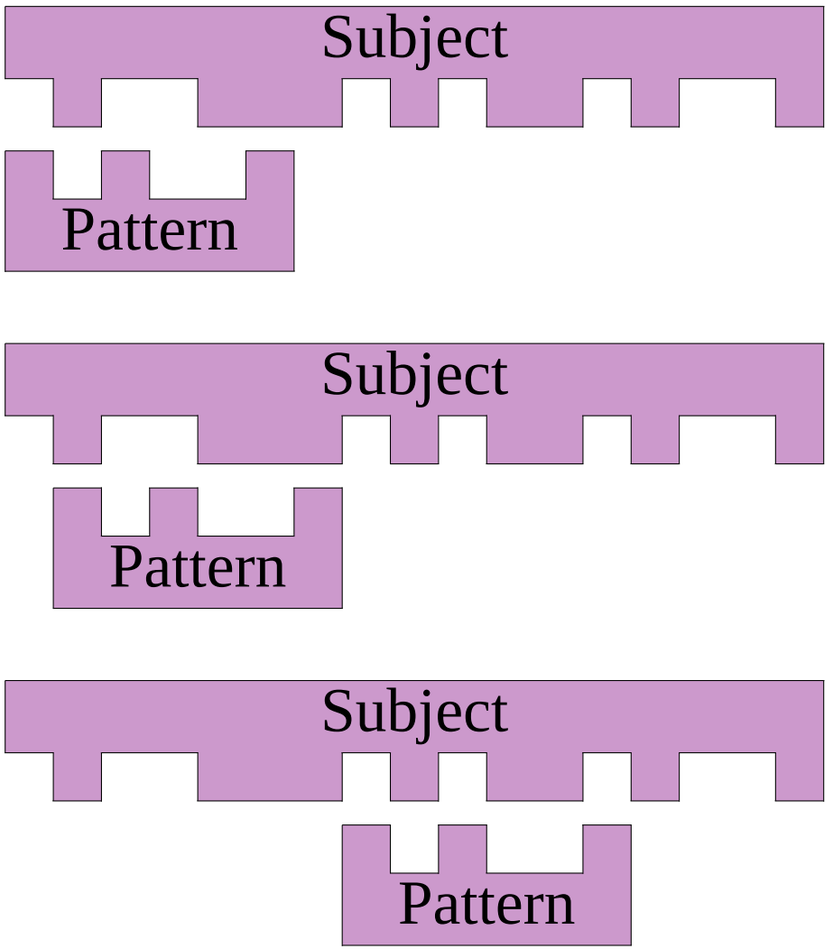
Horrible warning

- Regular expressions are very rarely defined precisely
  - Modern tendency to conflate specifications and users' guides
- The extensions are often mathematically inconsistent
  - So each is different and there are obscure "*gotchas*"
- They usually make the syntax fairly clear
  - The semantics (what it actually means) is often ambiguous
  - So you get no error, but it may misbehave
- This applies to at least POSIX, Perl and Python
  - As well as the previous Unix *egrep* etc.
- `KISS` – `Keep It Simple, Stupid`
  - Do that, and you will rarely have much trouble
  - Start to be clever, and you will shoot yourself in the foot
- This course has some guidelines on how to use them safely

Example of issues

- `a*` matches any number of `a`s, including none
  - No problem if anchored (i.e. starts at beginning)
  - But what if it isn't? And you want all matches?
- Replace each longest match by `x` in `1aa2aaa3`
  - Is it `1x2x3` or `x1x2x3x` or `x1x2x3` or ...
- So far I have found two of the above
  - As well as failures due to lack of space
  - Perl and Java produce `x1xx2xx3x` - yes, really!
- POSIX specifies `1x2x3` (following early System V?)
  - But *nothing* seems to conform to POSIX!
- Please tell me of any other variations ...

Matching a pattern within a string



Regular expression basics (1)

- Regular expressions deal with *characters*
- Modern implementations often support Unicode
  - PCRE supports wide characters using UTF-8 strings
- Most characters in a regex pattern stand for themselves
- However, the following are special *metacharacters*

```
* + ? ^ $ . ( [ { | \
```

- To make a metacharacter literal, precede it with `\`

```
cam\.ac\.uk
price: \$99.99
route: A1\(M\) north
```

- Remember to do that to the metacharacter `\`, too!
  - Generally, type `\\` to get `\`

Regular expression basics (2)

- Sometimes `\` before any non-alphanumeric makes it literal
  E.g. `\<\>` is the same as `<>`
  This is true for PCRE, Python, Perl and Java
- It is not true in some other implementations
  For example, `\<` and `\>` are metasequences in *egrep*
  (In fact, *egrep* is eccentric in several ways)

- Perl, PCRE, Java (not Python) have a literal substring facility
  `\Q...\E` quotes the text (i.e. ignores metacharacters)

```
\Qcam.ac.uk\E
\Qprice: $99.99\E
route: \QA1(M)\E north
```

- Particularly useful if you are matching unknown literal text
  For example, the contents of a Perl variable

Encoding problem characters (1)

- There is **immense** variation here
  - Even between two programs that seem to be the same
  - Following are most common conventions only
- Generally not part of regular expressions as such
  - String/character escapes are usually in **base** language
  - You must check your program/language for ‘gotchas’
- Escapes converted to characters during initial parse
  - But may **also** be accepted in regular expression syntax
  - Not always compatibly – `\b` is usually different
- Keep it very simple and you can ignore most of that
  - Main trick is to double up `\` to pass it through
  - E.g. `\\t` will usually reach the RE engine as `\t`
- Python raw strings make this easier
  - The handout includes some details and examples

Summary of Python string literals

- String literals are <prefix>'string' or <prefix>"string"
- Prefix letter `N`, `u` or `U` says it is a Unicode string

  `\N{name}` is character `name` in Unicode database
  `\uxxxx` is a 16-bit Unicode character
  `\Uxxxxxxxx` is a 32-bit Unicode character

- Prefix letter `r` or `R` says don't interpret `\` escapes
  The `\t` in `r'A\tB'` is really `\` and `t` and not tab
  Very useful for regular expression patterns!

- Prefix letter `ur` (either case) interprets only `\u` and `\U`
  All other backslashes are left alone

Python example

- Try the following in Python:

```
from re import match
x = 'A\tB';    y = 'A\\tB';    z = r'A\tB'
print x
print y
print z
print repr(x), repr(y), repr(z)
print match(x,x), match(x,y), match(x,z)
print match(y,x), match(y,y), match(y,z)
print match(z,x), match(z,y), match(z,z)
```

- Now, why are all three equivalent as patterns?
  - But only the decoded string is matched as a subject?
- Because the RE engine interprets `\t` as TAB
  - And only the decoded one has a real TAB in it

Encoding problem characters (2)

- C-style `\t`, `\n`, `\r`, `\f`, `\v` are usually accepted
  `\a`, `\e`, `\b` **sometimes** do what you expect
- `\`*ddd* specifies an octal number
  A maximum of three octal digits (always provide three)
  Example: `\133` (= 91) encodes [in ASCII/Unicode
- The null byte (`\000` or `\0`) may be allowed
  It may be special in **very** strange ways
- Octal is not recommended because
  People don't use octal any more (Unicode is defined in hex)
  Low numbers (1 to 99) are used for other purposes
  E.g. Python *groups* and Perl/PCRE *back references*

Encoding problem characters (3)

- `\x`*dd* specifies a hex number
  - Often limited to the range `\x00` to `\xff`
  - Example: `\x5B` (= 91) encodes [in ASCII/Unicode
- Good programs ignore case, some demand one or other
- Watch out for unexpected concatenation or termination
  - Is `\x5BCD` the same as `[CD` or Unicode 23501?
- Perl and PCRE users should check `\x{`*dd...*`}`
  - Mainly useful for 16+ bit Unicode characters
- Python Unicode is cleaner but very different (see handout)
  - Java is different yet again

Encoding non-ASCII characters

- Unicode / ISO 10646 is how you do this nowadays
  - Python, Perl, PCRE and Java all support Unicode
  - Both in strings and in regular expressions
  - But in different, very incompatible ways ...
- In **some** Unices, *egrep* etc. support Unicode
  - May well be true in Microsoft systems, too
- You must check with your program/language/system
  - Unicode may not behave the way you think it does
  - Single glyphs may have multiple representations
  - Not covered further in this course
- Ask me for help if you need to internationalise
  - I.e. to support languages other than USA English
  - It is **not** just a matter of changing to Unicode

Matching single characters (1)

- The dot metacharacter matches any character except newline

  `b..k` matches `buck`, `bunk`, `back`, `bark`, ...

- To make it match newline in Python, use the `DOTALL` flag
  To do so in Perl, set single-line mode with `\s`
- [...] matches one character from the set in [] (a character class)

  `reali[sz]e` matches `realise` or `realize`

- Dot and most other metacharacters are not special in classes

  `[.*]` matches `.` or `*`

- Hyphens are used for ranges

  `[0-9a-f]` matches a lower case hexadecimal digit
  Use only within digits, one case of letters or numerically
  `[3-7]` or `[I-N]` or `[\x20-\x7e]`

Matching single characters (2)

- If needed as a literal, a hyphen can be given first, or escaped

  `[-a-zA-Z0-9]` or `[a-z\-A-Z0-9]`

- ^ at the start of a character class inverts it

  `[^0-9]` matches any non-digit

- Note that a negated class still matches one character

  `b[^u][^c]k` matches `bark`, `beak`, `b34k`, ...

  But not `bok` (or `bunk`, `back` or `buck`)

- If needed as a literal, a ^ can be placed later or escaped

  `[a-z^]` or `[\^a-z]`

Character class shorthands (1)

`\d`	any digit	`\D`	any non-digit
`\s`	any whitespace char	`\S`	any non-whitespace
`\w`	any “word” char	`\W`	any non-“word” char

- The “word” characters are letters, digits, and underscore
- Let’s match a typical (old) Cambridge postcode

```
CB\d\s\d[A-Z][A-Z]
```

- This matches `CB4 2DB`, `CB2 3QH`, etc.
- Why use `\s` instead of an actual space?
  - It makes it clear how many space characters are being matched
  - We’ll see later there’s an option for ignoring literal white space
- But note that `\s` matches more than just the space character
  - Tab, newline, carriage return, etc. are also matched by `\s`
- Warning: the categories may be ASCII, locale or Unicode
  - Check your language, and there may be options, too

Character class shorthands (2)

- The shorthands can be used in bracketed character classes

  `[-+\d]` matches a minus, a plus, or a digit
  `[\d\s]` matches a digit or a whitespace character

- Double negatives can be hard to grasp

  `[^\W_]` matches a letter or a digit, but not an underscore

- Read "... *or* ..." for a positive character class
  Start with an empty set
  Each item inside [] *adds* to the set of characters

- Read "*not* ... *and not* ..." for a negative character class
  Start with a set containing all possible characters
  Each item inside [^] *subtracts* from the set of characters

POSIX character classes (1)

- They originated in HP-UX *tr*, if I recall
  - Most utilities for most Unices now support them
- A name inside [: :] can be used within a character class

  `[01[:alpha:]%]` matches `0`, `1`, any alpha character, or `%`
- Their availability is still patchy
  - Some systems/programs may use them incorrectly
- Perl and PCRE support them in their regular expressions
  - Python does **not** support them
  - Java uses the syntax `\p{name}`
- Stand-alone POSIX classes are not usually recognised

  `[:alpha:]` is equivalent to `[:ahlp]`
- The categories will generally depend on the locale
  - But not in Java, apparently ...

POSIX character classes (2)

- The available names are

`alnum`	letters and digits	`lower`	lower case letters
`alpha`	letters	`print`	printing, including space
`ascii`	characters 0–127	`punct`	printing, not alphanumeric
`blank`	space or tab only	`space`	white space
`cntrl`	control characters	`upper`	upper case letters
`digit`	decimal digits	`word`	“word” characters
`graph`	printing, not space	`xdigit`	hexadecimal digits

- **word** is a Perl extension; **blank** is a GNU extension
- Another Perl extension is negation

  `[12[:^digit:]]` is a complicated version of `[^03-9]`

- See the handout for Unicode properties

  Plus Java’s zoo of property criteria

Unicode properties in Perl and PCRE

`\p{x}`	character with Unicode property `x`
`\P{x}`	character without Unicode property `x`

- These are some typical Unicode properties

`L`	Letter	`Nd`	Decimal number
`Ll`	Lower case letter	`Nl`	Letter number
`Lt`	Title case letter	`P`	Punctuation
`Lu`	Upper case letter	`S`	Symbol
`M`	Mark	`Z`	Separator
`N`	Number	`Zs`	Space separator

- Script names such as `Arabic` or `Greek` can also be used
  - Perl supports a lot more properties; PCRE just the basics
  - Braces can be omitted for single-letter properties
- `\p` and `\P` can be used in character classes
- `\X` matches a non-mark character plus following mark(s)
  - `\X` just matches `X` in a character class

Unicode and Other Properties in Java

- Java typically uses `\p{name}` and `\P{name}`
  - Whether ASCII/POSIX, Java or Unicode depends on `name`
  - Simple categories (`\w` etc.) are always ASCII
- Bizarrely, the locale seems never to be considered
  - Which is a serious trap if you use locales in Java
- The Java criterion seems to be an old Unicode
- `&&` stands for intersection in character classes
  - This is any Unicode letter **not** in upper case

```
[\p{L}&&[^\p{Lu}]]
```

- Mixing category criteria is asking for trouble

```
[\p{L}&&[^\p{javaUpperCase}]]
```

Repetition (1)

- Curly brackets (aka braces) define *quantifiers*

`x{4}`	matches	`xxxx`
`x{1,3}`	matches	`x`, `xx`, or `xxx`
`x{2,}`	matches	`xx`, `xxx`, `xxxx`, `xxxxx`, ... any number
`x{,2}`	matches	nothing, `x` or `xx` in Python
`x{,2}`	matches	`x{,2}` in Perl and PCRE (not recognized)
`x{,2}`	is an error in Java	

- There are shorthands for the common repetitions

`x+`	means	`x{1,}`	(one or more `x`'s)
`\d*`	means	`\d{0,}`	(zero or more digits)
`[sz]?`	means	`[sz]{0,1}`	(zero or one `s` or `z`, i.e. optional)

- Quantifiers are "greedy" – adding ? makes them "ungreedy"

`ab[xyz]+`	with	**`abzxyycd`**	matches	`abzxyy`
`ab[xyz]+?`	with	**`abzxyycd`**	matches	`abz`

Repetition (2)

- Using the correct greediness can be important when there is more than one way to match a pattern
- Comments in the C language are of the form

```
/* This is a comment */
```

- The most naive pattern is (each item spaced out)

```
/  \*  .*  \*  /
```

- But it fails on this line

```
/* First */  a = b;  /* Second */
```

- Ungreediness solves the problem

```
/  \*  .*?  \*  /
```

- Why is this alternative approach also flawed?

```
/  \*  [^*]*  \*  /
```

- It fails on this line

```
/* This is an *important* comment */
```

Simple assertions (1)

- An assertion does not match any characters in the subject
  - Instead, it requires a condition to be true
- These are the simple assertions
  - `^` is true at the start of the subject
  - `$` is true at the end (more on this later)
- Match the leading path in a Unix filename such as `/a/b/c`

```
^.*/
```

- Match the trailing component of a Unix filename

```
[^/]*$
```

- The above pattern cannot fail, but is inefficient
  - It backtracks over 40 times for `/usr/local/bin/perl`
  - We will come back to this example

Simple assertions (2)

- For PCRE, Python, Perl and Java at least:

  `\b` is true at a word boundary
  `\B` is true not at a word boundary

- **This** `\b` is different from the C-style escape!
- In some utilities in some Unices, use `\<`word`\>`
  Check before using it, as it is a bit unreliable
- The start and end of the subject are treated as word boundaries

`\bcat\b`	matches **`cat`** and `the` **`cat`** `sat`
	does not match `cattle` or `scat`
`\Bcat\B`	matches `s`**`cat`**`ter`
	does not match `cat`, `cattle` or `scat`
`\bcat\B`	matches **`cat`**`tle`
	does not match `cat`, `scatter` or `scat`

Simple assertions (3)

- Pick an Email address out of other text

```
\b[^@\s]+@[a-z0-9.-]+\b
```

- This is a very crude solution
  - It does not do proper checks on the two parts of the address
- But it might be good enough if you know what is in your data
  - A regex that fully checks an Email address is very long indeed

Assertions inside qualifiers etc.

- In PCRE etc., assertions are dynamic
  - The following will do what you want – nowadays

```
(^\d+:)? *(\w+) *= *(\S+)
\S+ *(#.*|$)
```

- You should avoid such uses when possible
- Mainly because they are tricky to get right
  - They will not work reliably on pre-POSIX RE engines
  - And may not work for all POSIX utilities, either
- The above are more cleanly written as

```
^(\d+:|.*) *(\w+) *= *(\S+)
\S+ *(#.*)?$
```

- Note that POSIX itself doesn't define `\b`
  - Utilities will **usually** treat it as BS (control-H)

Text containing newline characters

- Most Unix utilities split the file into lines first
  - The subject is each line, without its newline
- But, in general, a subject may include newlines
  - `$` is true at the end **and** before a terminating newline
- Python `\A` and `\Z` match absolute start and end
  - They never recognize any newlines
- In Perl, PCRE and Java, that is `\A` and `\z`
  - `\Z` in those will also match before terminating newline
- `^` and `$` behave differently in “multiline mode”
  - Multiline mode is set by the `MULTILINE` flag in Python
  - Java also uses `MULTILINE`, but Perl uses `/m`
- In that case `^` matches after internal newlines
  - `$` matches before internal newlines
  - `^cat$` matches `dog\n`**`cat`**`\nrabbit` in multiline mode

Alternation

- Vertical bar separates alternatives
  - Any number are permitted

  `horse|donkey|mule` matches `horse`, `donkey`, or `mule`
- Parentheses are used to localize alternation
  - An empty alternative is permitted

  `cat(aract|erpillar|)`
  matches `cataract`, `caterpillar`, or `cat`
- Parentheses may be nested to any depth

  `J(anuary|u(ne|ly))` matches `January`, `June`, or `July`
- Parentheses may be followed by quantifiers

  `M(iss){2}ippi` matches `Mississippi`

Further examples (2)

- Match an optionally signed integer or decimal number

```
[+-]?[0-9]*\.?[0-9]*
```

- This matches `42`, `-272.37`, and even `.999`
- But what is wrong with it?
- It is *all* optional! So it will match `.` or the null string
- We have to use two alternatives if we want to allow `.999`

```
[+-]?(\d+(\.\d*)?|\.\d+)
```

Further examples (3)

- Match an HTML tag embedded in text

```
<[^>]+>
```

- This “obvious” pattern fails on tags such as this

```
<input name=dir value=">">
```

- HTML allows single or double quotes
  But embedded quotes in strings are not permitted
- Thus, we can use this pattern (ignoring white space)

```
< ( "[^"]*" | '[^']*' | [^'"] )* >
```

- The more advanced lecture considers the efficiency of this

Anchored patterns (1)

- A pattern is *anchored* if it can match only at the start
  - There is as much variation as for problem characters
- For safety, `^` should be at start (and `$` at end)
  - Except in a character class, or if escaped
  - Best to enclose alternations in parentheses

`^In the beginning`	is anchored
`^(soup|fish)`	is anchored

- The following will work in most modern systems
  - Older (non-POSIX) RE engines may do strange things
  - The same applies to some of the POSIX utilities

`^soup|^fish`	is anchored
`^entree|dessert`	isn’t anchored
`entree|^dessert`	isn’t anchored
`(^spoom|^buffet)`	is anchored

Anchored patterns (2)

- Is this pattern anchored?

  `.*something`

  It is sometimes anchored and sometimes not

- The dot metacharacter does not match a newline by default
  This pattern matches `other\n`**or something** `else`
  The match is not at the start
- In “dotall mode”, however, it *is* anchored
  Dot matches any character whatsoever in that mode
  “Dotall mode” was described earlier

- `match` forces anchoring in Python – `search` doesn’t

Capturing parts of the subject (1)

- Parentheses also define *groups* or *capturing subpatterns*
  - Opening parentheses are counted from the left starting at 1
- Example

  Pattern: `the\s((red|white)\s(king|queen))`
  Subject: `Here is the red king.`
  Result: Whole match = `the red king`
  Group 1 = `red king`
  Group 2 = `red`
  Group 3 = `king`

- Use nested capturing parentheses carefully
  - It is very easy to confuse yourself
  - Even simple changes will often change the group numbers
- We can avoid that the changing number problem
- That is covered shortly, but ignore it for now

Using the capturing parts

- Python groups are referred to as `\n`, with n = 1...99
- In `sub` only, `\g<n>` is equivalent to `\n`
  `\g<0>` stands for the whole match
- In Perl, the results of capturing are in `$1`, `$2`, etc.
  The entire matched string is in `$&` (*not* `$0`)
- Java is entirely different – see the `Matcher` class
- Other implementations provide similar facilities
  Exim, for example, is the same, but uses `$0` instead of `$&`
- Some implementations support *named* parentheses
  PCRE, Python, and Perl 5.10 are examples
  Those are covered in the more advanced lecture

Extended parenthesis features

- Conflating grouping and capturing can be a nuisance
- Perl/Python introduced an *extended parenthesis* feature
- An extended parenthesis starts with `(?`
  `(?:...)` parentheses are simple grouping (non-capturing)
  There are also a number of other extensions that use `(?`
- Compare

```
the\s((red|white)\s(king|queen))
the\s((?:red|white)\s(?:king|queen))
```

- There is only one capturing subpattern in the second version
  Using `(?:...)` is tedious but is **strongly** recommended
  Many examples here use `(...)` for clarity only

Capturing parts of the subject (2)

- Example pattern and subject:

```
(\w+):\s*(\w+(?:/\w+)*)\s+(\.*)
```

```
Joan:  jumping/alone     Nobody's with me
```

Group 1 = `Joan:`
Group 2 = `jumping/alone`
Group 3 = `Nobody's with me`

- And we could extend it without changing that:

```
(\w+):\s*(\w+(?:/(?:\w+|\.|\.\.))*)\s+(\.*)
```

Capturing parts of the subject (3)

- Avoid capturing within alternations or quantifiers
  - Exactly **what** does `(\w+|\w+(/\w+)+)*` capture?
  - Or even `\w+|\w+(/\w+)+`
- You need to know the implementation to even guess
  - It might vary with the version as changes are made
  - It **will** vary with the context in deceptive ways
- Note that I have been sneaky with the latter form
  - Can **you** see why it won't work?
- If you can't, don't worry about it - just don't do it
  - It is explained in the more advanced lecture
  - Look for a slide including `catastrophe`

Further examples (4)

- Extract the path and the final component of a Unix filename

```
^(.*/)([^/]*)$
```

- This is also the fast way just to extract the final component
- Why would this pattern also work?

```
^(.*/)(.*)$
```

- To handle names that do not contain a slash you might try

```
^(.*/)?([^/]*)$
```

- But that will also match the empty string
- The extracts end up in different parentheses if you use

```
^(?: (.*/)([^/]*) | ()([^/]+) )$
```

- The problem can be solved, but we need to learn more
  (Use conditionals or named parentheses)

Changing processing options in Python

- The main functions or methods have an optional flag argument
  `compile`, `search`, `match`, `findall`, `finditer`
- The following can be combined by ORing, as in `X|I`

`IGNORECASE (or I)`	sets case-independent matching
`VERBOSE (or X)`	ignores literal white space
`DOTALL (or S)`	`.` also matches a newline
`MULTILINE (or M)`	`^` and `$` match internal newlines
`LOCALE (or L)`	char. classes use the current locale
`UNICODE (or U)`	char. classes use the Unicode database

- Character class default if not `L` or `U` is ASCII
- `VERBOSE` can be used to make a regex pattern more readable
  It also treats an unescaped # as introducing a comment
- You can also set them within a RE by `(?<letters>...)`
  For example `(?x)` allows you to use space as layout

Changing processing options in Perl

- These strings change processing options
  - `(?i)` sets case-independent matching
  - `(?x)` ignores subsequent literal white space
  - `(?s)` `.` matches a newline as well as any other character
  - `(?m)` `^` and `$` match after/before internal newlines
- Perl uses rather confusing terminology
  - `(?s)` sets “single-line mode”) but you can set
  - `(?m)` sets “multi-line mode”) both at once!
- Changes last till the end of any enclosing parentheses
  - `(a(?i)b)c` matches `abc` or `aBc` but nothing else
  - `(a(?i)b|c)` matches `ab`, `aB`, `c`, or `C`
- A hyphen is used for negation, and options can be combined
  - `(?x-i)` ignores white space, sets case-dependence
- `(?x)` is useful to make a regex pattern more readable

Changing processing options in Java

- The `compile` method has an optional flag argument
  Just like Python, but with different meanings
- The following can be combined by ORing

`CASE_INSENSITIVE`	sets case-independent matching
`COMMENTS`	ignores literal white space
`DOTALL`	`.` also matches a newline
`MULTILINE`	`^` and `$` match internal newlines
`LITERAL`	Metacharacters are not special
`UNICODE_CASE`	Case independence uses Unicode
`CANON_EQ`	Use Unicode canonical equivalence
`UNIX_LINES`	`\n` is only line terminator recognised

  No, I don't know which characters `LITERAL` affects
  Ask me offline what canonical equivalence means

- `(?x)` etc. are allowed, exactly as in Python and Perl

Optimisation (1)

- There's more than one way to tackle any problem
- Don't forget that array operations are much quicker
  - And the 'basic' string ones usually are, too
- But, if you think you need `strtok`, you need REs
  - Similarly when you start to write complex logic
- As always, if you have a problem, stand back and think
  - You may not be using the best tool for the job
  - Or may be tackling it the wrong way
- And, if you have problems, please ask for help

Diversion – multiple stages (1)

- Are you going cross-eyed? I did when editing these slides
- A real expert can write an efficient, complex, REs
  - My experience is that I am not that good
  - I know the theory quite well, and use them quite often
- Biggest problem is in reliability of error detection
  - Complex REs will match some strings that are wrong
  - And will you spot that? It isn't easy even in simple ones
- Will this recognise Fortran numeric data and nothing else?

```
(?:[+-] *)?(?:\d+\.?|\d?\.\d+) ...
... (?:[EeDd] *(?:[+-] *)?\d+)?
```

- Answer: it will, though that fact may surprise you
  - And those spaces are only spaces – using `\s` is wrong
- Matching complex text in stages can be a simpler solution
  - When it helps, it is as fast and much clearer

Diversion – multiple stages (2)

- Consider checking for gross errors (e.g. invalid chars)
  - Either using simple REs or basic string operations

```
^[\s\x20-\x7e]*$
^[\s\w/\.+-]*$
```

  - Subsequent regular expressions can then be simplified
- Very often can split into tokens and then unpick those
  - Words may be separated by white space and punctuation
  - Data formats are often words, numbers and other chars
- Fortran is statement-based, and so are many data formats
  - You can split into statements, and **then** decode each one
- First stage does continuation, strings, escapes and comments
  - Second stage classifies the statement into classes
  - Then each class is decoded with its own expression

Diversion – multiple stages (3)

- Can often avoid using complex expressions on huge data
  - The initial splitting can use very simple ones
  - It is much easier to optimise simpler expressions
- Don’t need to use the advanced features for efficiency
  - Easier to code and debug, and more portable
- Python users should check `findall` and `split`
  - I have speeded up by 10 times by using `findall`
  - I am sure Perl and Java have comparable facilities
- Perl users can avoid embedding code in expressions
- As always, there are cases when it won’t work
  - Then the above (and following) techniques are needed
- Use whatever you find easiest to work with
  - Never write ‘clever’ code unless you really need to

Optimisation (2)

- Primary objective is to minimise the number of steps
  - Only secondarily worry about using faster primitives
- Check the most likely possibilities first
- Abandon the check as soon as it is hopeless
- Above all, don’t wander around everywhere deciding
  - I.e. don’t backtrack more than you have to
- The rest of this course is largely how to avoid this
  - Assuming the ‘conventional’ order of execution

Stopping backtracking (not in Python)

- Consider this pattern for matching an Email domain

```
^x\d*y\.example\.com$
```

- Matching engines are not clever; they try every alternative
  - There is no problem if the match succeeds
  - Backtracking occurs when the match fails
- Consider: `x123456789z.example.com`
- Atomic grouping stops backtracking
  - Available in PCRE, Perl and Java

```
^x(?>\d*)y\.example\.com$
```

- Such a group matches what a standalone pattern would match
- For single items, a *possessive quantifier* does the same thing
  - Originally from Java; then in PCRE; will be in Perl 5.10

```
^x\d*+y\.example\.com$
```

Generalized assertions: lookahead (1)

- Forward assertions look ahead

  `foo(?=bar)` matches `foo` if followed by `bar`
  `foo(?!bar)` matches `foo` if not followed by `bar`

- Example

  ```
  ^x(?!999)\d{3}
  ```

- This matches `x123`, `x456`, etc., but not `x999`
- Another example

  `(?=Jeffrey)Jeff` matches **`Jeff`**`rey Friedl`
  does not match `Thomas Jefferson`

- Note that `Jeff(?=rey)` has the same effect
- What does `Jeff(?=Jeffrey)` match?

  Answer: **`Jeff`**`Jeffrey`

Generalized assertions: lookahead (2)

- Lookahead does not “consume” text
- Forward assertions are not backward assertions

  `(?=bar)foo` does *not* match `foo` if preceded by `bar`
- `(?=bar)foo` cannot succeed: it always fails
- You can force an alternative to fail with `(?!)`

  See later about conditionals – useful in a branch

Generalized assertions: lookahead (3)

- Can be used when side-effects are available, as in Perl
- You are advised **not** to embed code in expressions
  Load gun, point at foot, pull trigger ... **BANG!**
  But there are occasions when it can be useful
- Consider the following program

```
"123" =~ m{ \d+ (?{ print "$‘<$&>$’  " })
          (?!) }x;
```

- The output is

```
<123>  <12>3  <1>23  1<23>  1<2>3  12<3>
```

- With an ungreedy quantifier the output is

```
<1>23  <12>3  <123>  1<2>3  1<23>  12<3>
```

Generalized assertions: lookbehind

- Backward assertions look behind

  `(?<=bar)foo` matches `foo` if preceded by `bar`
  `(?<!bar)foo` matches `foo` if not preceded by `bar`

- Backward assertions must be of fixed length
  In PCRE only, each branch may be a different length
- We want to match any domain ending in `.xxx.com`

  `^.*\.xxx\.com$`

- That is usually quite fast, but the following is faster

  `^(?>.*)(?<=\.xxx\.com)$`

- The following is **not** a good idea
  `^.*(?<=.xxx.com)$`
- It is much better to match in two stages (see later)

Conditional subpatterns

- A group id may be followed by one or two alternatives

  `(?(`*group*`)`*yes-pattern*`)`
  `(?(`*group*`)`*yes-pattern*`|`*no-pattern*`)`

- The condition matches if the group exists

  The id may be a group number or group name

```
(?x) ( \( )?  [^()]+  (?(1) \) )
```

- This matches an optionally parenthesized string

Assertions in conditionals

- This is available in PCRE and Perl
  - It is not available in Python or Java
- The condition can be an assertion

```
(?xi)  (?(?=[^a-z]*[a-z])
       \d{2}-[a-z]{3}-\d{2} |
       \d{2}-\d{2}-\d{2} )
```

- This matches dates of the form `12-Feb-07` or `12-02-07`
- The assertion can be forward or backward, positive or negative

Further examples (5)

- Recall the file name matching example

```
^(?: (.*/)([^/]*) | ()([^/]+) )$
```

- This allows for names with and without slashes
  - We had to use top-level alternatives with different parentheses
  - We would like to use the same capturing parentheses
- A conditional test does the trick

```
^ (.*/)? ( (?(1) [^/]* | [^/]+) ) $
```

- There is one subtle difference
  - The first capturing parentheses can be unset
  - Use an explicit null alternative to avoid this

```
^ (.*/|) ( (?(2) [^/]* | [^/]+) ) $
```

- The required extracts are now always in **$1** and **$2**

Back references

- A back reference matches exactly what the subpattern matched
  `(woof|miaow),\s\1` matches `woof, woof`
  or `miaow, miaow`
  does not match `woof, miaow`
- How to detect repetition typos such as “the the”
  `\b([a-z]+)\s\1\b`
- See the handout for a Perl program to do this

- In Python, they are like the first form in substitution
- `\1` to `\9` and`\10` to `\99` are back references
- `\0`, `\0d...` and `\ddd` are octal escapes
- Both are characters in the `[...]` of a character class

Back references in Perl 5.10 & PCRE 7.0

- Backslash followed by a number can be ambiguous
  - Confusion with octal character definitions
  - What if a digit follows?
- `\g` followed by a number is always a back reference
  - Braces can be used to delimit the number

  `(123|456)\g{1}0` matches `1231230` or `4564560`
- A negative number can be used for a relative back reference
  - `\g{-1}` refers to the most recently started group
  - `\g{-2}` refers to the next most recently started group, etc
  - This is helpful in the middle of long patterns

  `stuff ... (123|456)\g{-1}0 ...`
- Most recently started is not always most recently ended

  `(A(12|34)B)\g{-1}` matches `A12B12` or `A34B34`

Back references in Java

```
\1 through \9 are always interpreted as back
references, and a larger number is accepted
as a back reference if at least that many
subexpressions exist at that point in the
regular expression, otherwise the parser will
drop digits until the number is smaller or
equal to the existing number of groups or it
is one digit.
```

- That is a **truly** revolting specification
- Add some parentheses and a much later digit disappears
  - I will explain at the end why this is so stupid
- What if there is still no match? I couldn’t see

A program to display repeated words

- Jeffrey Friedl's complete Perl script

```
$/ = ".\n";
 while (<>) {
   next if
     !s/\b([a-z]+)((?:\s|<[^>]+>)+)(\1\b)
       /\e[7m$1\e[m$2\e[7m$3\e[m/igx;
   s/^(?:[^\e]*\n)+//mg;  # Remove unmarked
   s/^/$ARGV: /mg;        # Add filename
   print;
 }
```

- You use it by `perl <script> <file>`

Subroutines and recursion

- "Subroutine" references are possible in PCRE
  - They will be available in Perl 5.10
  - They are not available in Python or Java

  `(woof|miaow),\s(?1)` *does* match `woof, miaow`

- Recursive use of this mechanism is permitted

```
(?x) ( \( ( [^()]++ | (?1) )* \) )
```

- This matches nested parenthesized strings like `(1*(2+3)-4)`
  - Note the use of a possessive quantifier for efficiency

- The following pattern matches palindromes!

```
(?i)^\W*(?:((.)\W*(?1)\W*\2|)|
  ((.)\W*(?3)\W*\4|\W*.\W*))\W*$
```

Recursion support in Perl before 5.10

- The “subroutine” features of PCRE are not available
- Interpolated code can be used to achieve the same effects
- Load gun, point at foot, pull trigger ... **BANG!**
- The following Perl pattern matches a parenthesized group

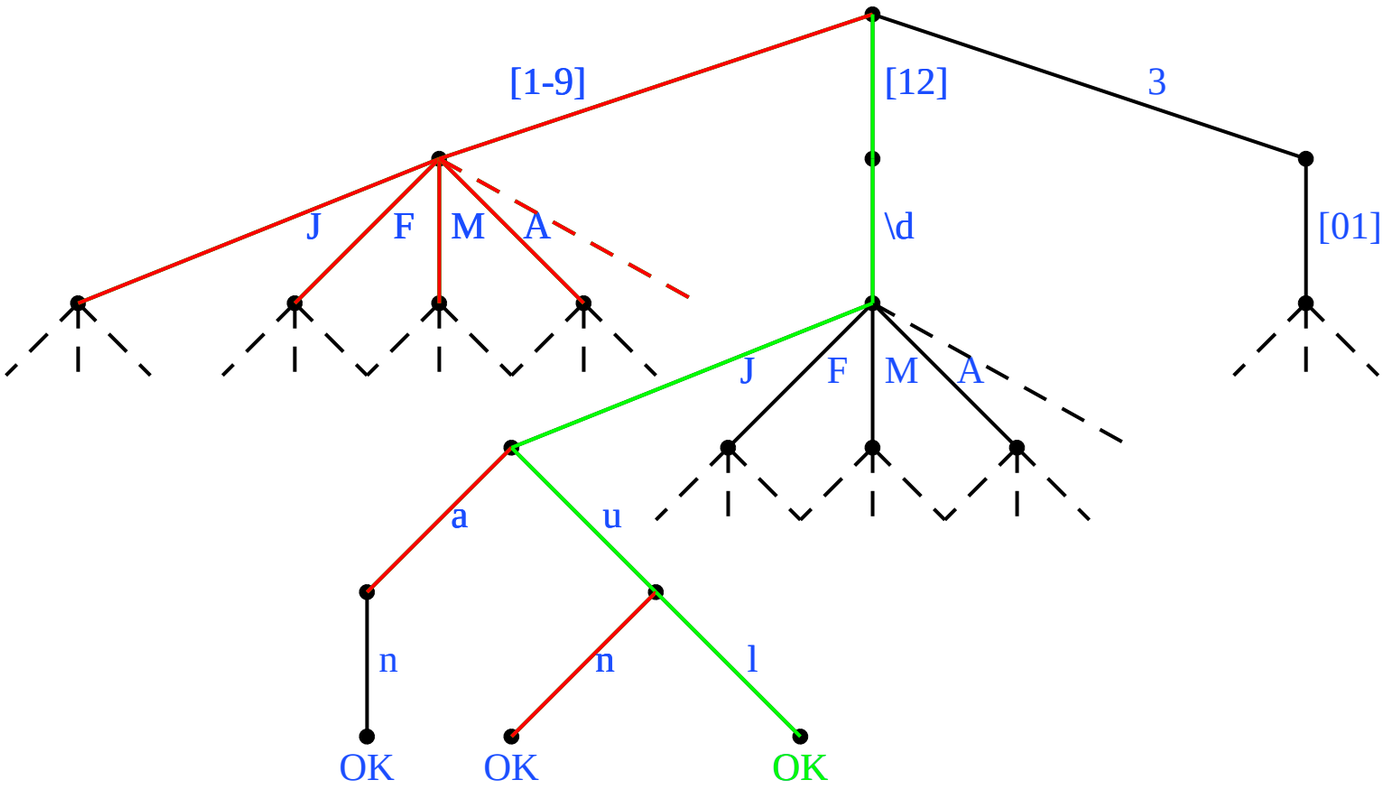
```
$re = qr{
   \(                      Opening parenthesis
   (?:                     Non-capturing group
      (?> [^()]+ )         Non-parentheses without backtracking
    |                      Or
      (??{ $re })          Parenthesized group
   )*                      Any number of either
   \)                      Closing parenthesis
}x;

$subject = "(1*(2+3)-4)";
if ($subject =~ /$re/) { ... }
```

Matching a pattern the Perl way (1)

Pattern:

```
^([1-9]|[12]\d|3[01])(J(an|u(n|l))|Feb|...)
```

Subject: `18Jul`



Matching a pattern the Perl way (2)

- A Perl-like engine returns the *first leftmost* match
  It searches the tree *depth first* ; Friedl calls this a
  "*Nondeterministic Finite Automaton*" (NFA) method
- See the end of the handout about the proper theory
  Friedl and this course are entirely about actual practice
- The order in which branches are checked depends on

  (1) The ordering of any alternatives) these are controlled
  (2) The greediness of any quantifiers) by the pattern writer
- Depth first is used by GNU Emacs, Java, *grep* (most versions), *less*, *more*, the .NET languages, PCRE, Perl, PHP, Apache, Python, Ruby, *sed* (most versions), *vi*

Matching a pattern the Perl way (3)

- Some patterns are nonsensical

  `(cat|catastrophe)` does not work as expected

- Why not?
- In Python (as in Perl), the following statement:

```
print re.sub(r'(cat|catastrophe)', \
    r'fiasco', 'catastrophe')
```

- Produces the following result:

```
fiascoastrophe
```

- A depth-first algorithm can capture substrings easily
  - It identifies just one path through the tree
  - The order is mostly what you expect, if not always
- Full access to both pattern and subject is needed throughout

POSIX Regular Expressions

- POSIX mandates that the *longest leftmost* match is found
  - It does not say how it is to be found
- A “traditional” NFA engine might not find the longest match
- A POSIX-compliant NFA engine must continue searching
  - All possible branches must be explored
  - This can soak up resources
- But:

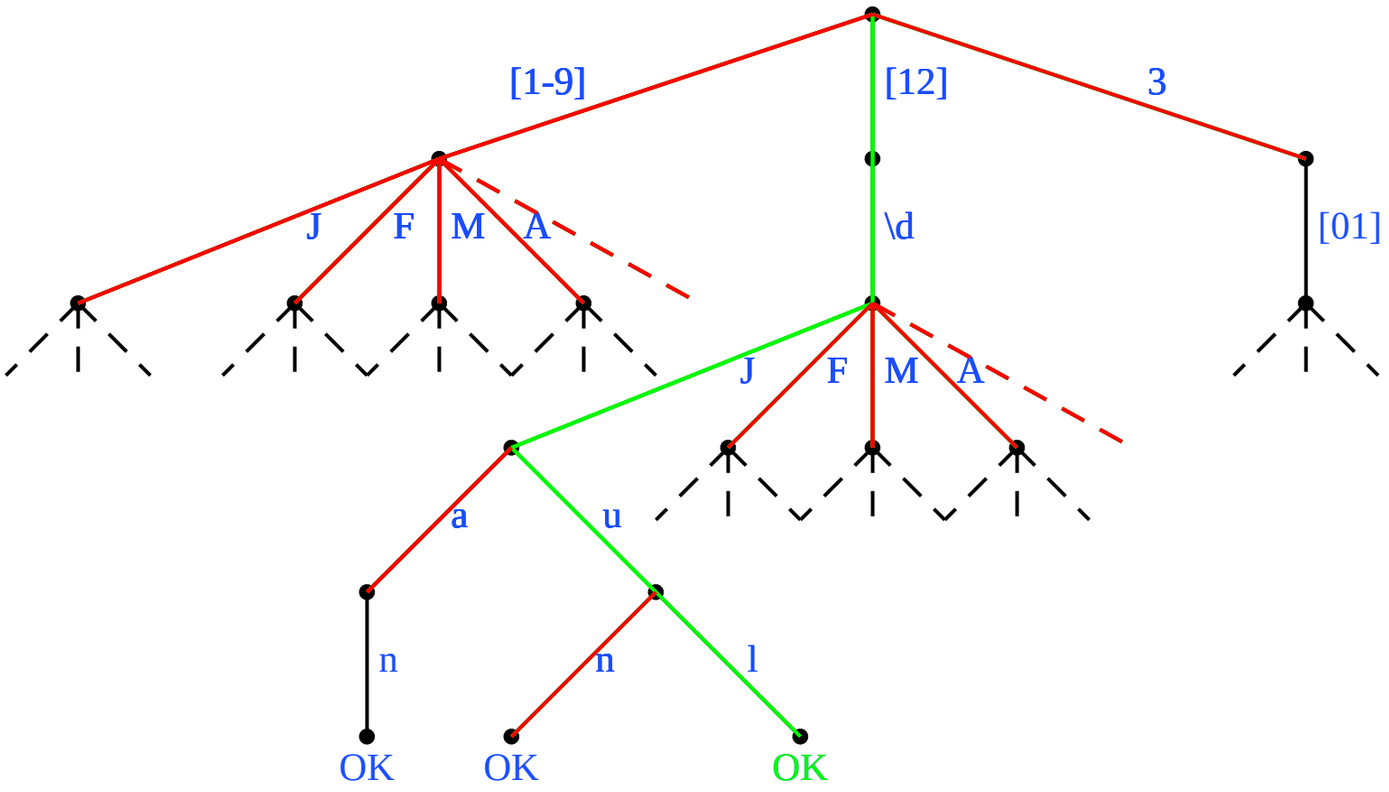
  `(cat|catastrophe)` works as you might expect
- A POSIX NFA engine is used by *mawk* and can be requested in GNU Emacs

Matching a pattern another way (1)

Pattern:

```
^([1-9]|[12]\d|3[01])(J(an|u(n|l))|Feb|...)
```

Subject: `18Jul`



Matching a pattern another way (2)

- The breadth-first way of searching never backtracks
  - Friedl calls this kind of algorithm a "*Deterministic Finite Automaton*" (DFA) method
  - Each subject character is inspected only once
- Neither greediness nor order of alternation matters
  - All matches are found (usually the longest is reported)
- Atomic grouping/possessive quantifiers may not be supported
- There is no capturing of substrings
- Breadth-first engines are used by *awk* (most versions), *egrep* (most versions), *flex*, *lex*, MySQL, Procmail
- Hybrid engines are used by GNU *awk*, GNU *grep/egrep*, Tcl
- PCRE contains an alternate breadth-first matching function
  - It is somewhat unlike other implementations
  - PH called it `pcre_dfa_exec` before he understood that

The C comment problem for breadth-first engines

- Ungreediness is useless because all matches are found
  ```
  /* First */  a = b;  /* Second */
  ```
- A breadth-first engine finds both these strings
  ```
  /* First */
  /* First */  a = b;  /* Second */
  ```
- In most applications, the longest match is preferred
- Why are there not three matches?
  Answer: matches are only ever reported at one position
- This is true for depth-first engines as well, of course
  Some applications have “global” options to continue matching
- For a breadth-first engine, the pattern has to be rewritten
  ```
  / \* ( [^*]+ | \*[^/] )* \* /
  ```
- Would this also work for a depth-first engine?
  Answer: no, try it on `/***/`

Performance of depth-first matching engines

- Think about non-matches as well as matches
  - For a non-match, all possible alternatives are tried
  - (With a breadth-first engine, they are always tried)
- Nested unlimited repeats are killers

```
(?x) ( [^"\d]+ | "[^"]+" )* \d
```

- This string is matched very quickly

```
some "quoted stuff" and a digit 4
```

- But it will take a very long time if you try

```
quite a lot of "quoted stuff" and no digit
```

  - Though not in Perl, for some reason!
- We describe how to tackle this problem later

The combinatorial explosion

- The pattern `^(a+)*` can match `aaa` in 8 different ways

0 outer repeats	matches an empty string
1 outer repeat `(a+)`	matches `a`, `aa`, or `aaa`
2 outer repeats `(a+)(a+)`	matches `a a`, `aa a`, `a aa`
3 outer repeats `(a+)(a+)(a+)`	matches `a a a`

- Each additional letter doubles the number of possibilities
  For `aaaaaaaa` (8 letters) there are 256 possibilities
- How can we avoid such disasters?
- Greediness/ungreediness does not help – why not?
  Answer: For a non-match, all possibilities are always tried
- Rewrite as `^(a+)?` when possible, as here
- Or use an atomic group or possessive quantifiers

```
(?x) (?> ( [^"\d]+ | "[^"]+" )* ) \d
(?x) ( [^"\d]++ | "[^"]++" )* \d
```

Nesting isn't just parentheses

- `K` levels of nesting on a sequence of length `N`
- Requires up to `N^K` backtracks on failure
- `((.*)*)*Z` obviously has `K = 3`
- Alternation can cause nesting

  `(.*|.*|.*)Z` also has `K = 3`
- Concatenation can cause nesting

  `(.*.*.*)Z` also has `K = 3`
- Generally, concatenation is handled best

  Then alternation, then parentheses
- But that is a question of how the code is optimised

  Each implementation may be different

Concatenation inside repeats

- We are talking about patterns like
  `( pat1 ? pat2 + pat3 * ) *`
- `(.*.*.*)*Z` is catastrophic on a mismatch
- `(?x) ( [^'"]* "[^"]*"+ )*` is fine (see later)
- Try to avoid patterns that can overlap
  - No possible overlap means minimal backtracking
- Remember that null matches count for overlaps!
  - `?`, `*` and `(...|)` can all be null
- In general, unambiguous patterns work well
  - Avoiding ambiguity is always worthwhile
  - Remember the `CAT`astrophe?

Further examples (6)

- Recall the HTML tag matching example

```
< ( "[^"]*" | '[^']*' | [^'"] )* >
```

- Each non-quoted character requires a group repeat
  - However, this modification can be disastrous

```
< ( "[^"]*" | '[^']*' | [^'"]+ )* >
```

- This is the fastest Pythonic version I can find

```
< ( [^'"]* |
    ( [^'"]* ( "[^"]*" | '[^']*' ) )+
      [^'"]* ) >
```

- It's not pretty, and rarely worth the effort
  - In Python, you should approach the problem differently
  - Only when this is a performance bottleneck, of course
- There are some hints of how in the handout

Further examples (7)

- Or you can use atomic grouping or possessive quantifiers
  They are not available in Python
- Atomic grouping

```
< ( "(?>[^"]*)" | '(?>[^']*)' |
    (?>[^'">]+) )* >
```

- Possessive quantifiers in PCRE, Perl 5.10 and Java

```
< ( "[^"]*+" | '[^']*+' | [^'">]++ )* >
```

- And none of the above expressions work on this:

```
This "<" and this ">" are not a HTML tag
```

- Extending them to do so is left as an exercise ...

Some example performance figures (1)

- *pcretest* (see later), Python, Perl and Java

```
<( "[^"]*" | '[^']*' | [^'"] )*>
<( "[^"]*" | '[^']*' | [^'"]+ )*>
<( [^"']* |
     ( [^"']* ( "[^"]*" | '[^']*' ) )+
         [^"']* )>
<( "(?>[^"]*)" | '(?>[^']*)'| (?>[^'"]+) )*>
<( "[^"]*+" | '[^']*+' | [^'"]++ )*>
```

- How long to report "no match" for a 20-character subject?

  All times in milliseconds

	pcre	Python	Perl	Java
Original	0.004	0.007	0.009	0.003
With the '+'	32.2	55.9	0.040	30.6
Concatenation	0.004	0.006	0.006	0.004
Atomic group	0.001		0.003	0.002
Poss. quants	0.001			0.002

Can we do better in Python?

- Yes, but only slightly and it gets a bit messy

	pcre	Python	Perl	Java
Original	0.004	0.007	0.009	0.003
Concatenation	0.004	0.006	0.006	0.004
These tweaks		0.004		
Atomic group	0.001		0.003	0.002
Poss. quants	0.001			0.002

- However, the HTML example is a bit deceptive
  - Finding the next a HTML tag may be a bottleneck
  - Decoding it when you have found one rarely is
- Generally, use multiple stages in Python
  - Concentrate on the bulk data operations
- Python is not suitable for all RE tasks
  - Well, actually, nor are PCRE, Perl or Java ...

Using `split` and `sub"/ in Python

- If you need the subpatterns, anyway, try this

```
pat1 = re.compile(r'^[^<]*<')
pat2 = re.compile( \
    '("[^"]*"|\'[^\']*\'|[^"\'>]+|.*)')
temp = pat1.match(text)
temp = pat2.split(text[temp.end():])
result = "".join(temp[0:-1])
```

- This is a bit perverse, but is quite fast

```
pat1 = re.compile(r'^[^<]*<')
pat2 = re.compile( \
    '"[^"]*"|\'[^\']*\'|[^"\'>]+')
temp1 = pat1.match(text)
temp1 = text[temp1.end():]
temp2 = pat2.sub("",temp1)
result = temp1[0:-len(temp2)]
```

Some example performance figures (2)

```
(?x) ( [^"\d]+ | "[^"]+" )* \d
(?x) (?x) ([^"\d]* |
     ( [^"\d]* "[^"]+" )+ [^"\d]* ) \d
(?x) (?> ( [^"\d]+ | "[^"]+" )* ) \d
(?x) ( [^"\d]++ | "[^"]++" )* \d
```

- How long to report “no match” for a 14-character subject?
  - All times in milliseconds

	pcre	pcre[*]	Python	Perl
Original	13	0.90	23.4	0.042
Concatenation	0.018	0.027	0.032	0.030
Atomic group	0.007	0.038		0.016
Poss. quants	0.010	0.022		

- [*] PCRE’s breadth-first engine (option `-dfa`)
  - This is not particularly fast
  - It is not a traditional finite state machine (DFA)

Testing PCRE regular expressions

- Using *pcretest* to debug and time regular expressions

```
$ pcretest -t
PCRE version 7.0 18-Dec-2006
  re> /(?x) ( [^"\d]+ | "[^"]+" )* \d/
Compile time 0.0026 milliseconds
data> the quick brown fox jumps over 42
Execute time 0.0007 milliseconds
 0: the quick brown fox jumps over 4
 1: the quick brown fox jumps over
data> the quick
Execute time 0.2953 milliseconds
No match
  re> /(?x) (?> ( [^"\d]+ | "[^"]+" )* ) \d/
Compile time 0.0031 milliseconds
data> the quick
Execute time 0.0052 milliseconds
No match
```

Timing in other languages

- Writing one makes a good second practical
  - Harder than “Hello, world!” but not very hard
- Please ask if you have difficulties
- Java 1.6 documentation on REs has some suitable code
  - It’s not hard to hack it to work under 1.5
- I wrote a fairly clean harness for Python
  - It is on the Web pages mentioned at the start
- My Perl test harness is too revolting to publish
  - I do not like Perl and avoid it if I can

Named parentheses (1)

- First made available in Python and PCRE using this syntax

```
(?x) (?P<day>\d\d)-
     (?P<month>[[:alpha:]]{3})-
     (?P<year>\d{4})
```

- The names are in addition to the numbers
- Back references use `(?P=`

  These two patterns are the same

```
(woof|miaow),\s\1
(?P<noise>woof|miaow),\s(?P=noise)
```

- A condition test can use a name in parentheses

  These two patterns are the same

```
(?x) (          \( )? [^()]+ (?(1) \) )
(?x) (?P<OPEN> \( )? [^()]+ (?(OPEN) \) )
```

Named parentheses (2)

- Perl 5.10 uses a different syntax, which PCRE also supports
  Python does not support this
- No `P` is used; the name may be in `<>` or single quotes

```
(?x) (?<day>\d\d)-
     (?<month>[[:alpha:]]{3})-
     (?'year'\d{4})
```

- Back references use `\k<name>` or `\k'name'`

```
(?<noise>woof|miaow),\s\k<noise>
```

- Conditionals use `(?(<name>)...` or `(?('name')...`

```
(?x) (?'OPEN' \( )? [^()]+ (?('OPEN') \) )
```

Named parentheses (3)

- In PCRE/Perl 5.10, recursive/subroutine calls can use names

```
(?x) ( \( ( (?>[^()]+) | (?1) )* \) )
(?x) (?<pn> \( ( (?>[^()]+) | (?&pn) )* \) )
```

- You can use a name more than once
  - In PCRE, the feature must be enabled with the J option

```
(?x)
(?<DN>Mon|Fri|Sun)(?:day)?|
(?<DN>Tue)(?:sday)?|
(?<DN>Wed)(?:nesday)?|
(?<DN>Thu)(?:rsday)?|
(?<DN>Sat)(?:urday)?
```

- DN contains the 3-letter name, whichever alternative matches
  - This facility could be confusing if not correctly used
- The above is probably the **only** safe way to use it!

Named parentheses (4)

- Named parentheses are also available in .NET languages
  The Perl <> is syntax used

```
(?x) (?<day>\d\d)-
     (?<month>[[:alpha:]]{3})-
     (?<year>\d{4})
```

- Numbers are also assigned, but not like Python/PCRE/Perl
  Non-named parentheses are numbered first

```
(\w) (?<num>\d+) (\s+)
1..1 3.........3 2...2
```

- Back references use \k, as in Perl

```
(?<noise>woof|miaow),\s\k<noise>
```

- Syntax for conditionals is as for Python

```
(?x) (?<OPEN> \( )? [^()]+ (?(OPEN) \) )
```

True regular expressions and their theory

- `Don't panic`
  - I am not going mathematical on you! – Well, not much
- It's worth knowing what true regular expressions are
  - Roughly why the difference can be important
  - And, most of all, when you need to use them!
- Most common use is for data input designs
  - Including passing formatted data between programs
- True REs greatly help debugging and reliability
  - You are more likely to spot accidental ambiguities
  - If an editor mangles a file, your code may detect it

Why do true regular expressions help?

- The mathematical key is context independence
  - That enables the streaming feature of breadth-first
  - One construct doesn't change the meaning of another
- Remember Java back references? Now consider

```
((lots\)(of\)(opaque))\(re)\(code)) ... \10
```

  Is that group 1 and digit 0 or group 10 and nothing?

- Parsing context-dependent code (old Fortran, C) is a **pain**
  - Which is why modern programming languages use REs
- They are closely related to Backus-Naur form (BNF)
  - The following is an easy-to-understand reference

```
http://en.wikipedia.org/wiki/Backus-Naur_form
```

- BNF is very useful when defining data formats

A very simple example

- Consider a vector input statement, accepting:

```
values = ( 1.23, 4.56e7,-89, ...
... 0.12e-3) Optional comment ;
```

- A reasonable format for a number is:

```
[+-]?\d+(\.\d+)(?:[EeDd][+-]?\d+)?
```

- Similarly the whole format is:

```
(?x) (\w+) \s* = \s*
\( \s* <number>
    (?: \s* , \s* <number> \s* )? \)
\s* \( \.\S \)? \s* ;
```

- Design formats like that, however you parse them
  I can assure you that it really does help

Original regular expressions

- You can use only the following:
  - Characters and character classes
  - Quantifiers
  - Alternation
  - Parenthesis grouping
- That's not how the textbooks describe them
  - The notation accepted by software and authors varies
  - Quantifiers are often only `?`, `+` and `*`
- These can be converted to an efficient form
  - Stick to these when defining data formats
  - Except when you really can't, of course
- There is a minimal summary of the theory in the handout

The theory in four slides (1)

- Regular automata arose out of language theory
  - Which is not about programming languages!
- A `language` is the set of matchable strings
  - They are about matching **only** – `yes` or `no`
  - Greediness and capturing are irrelevant
- All patterns are independent of their context
  - No backtracking control, atomicity, testing etc.
- NFAs and DFAs are functionally identical
  - Unlike in the extended expressions described above
- They have many desirable properties
  - Not always shared by extended ones, as above

The theory in four slides (2)

- A regular expression is the text notation for a `NFA`
  - So-called `Nondeterministic Finite Automaton`
  - There is nothing non-deterministic about them
- True non-deterministic automata are very different
  - Nothing to do with regular expressions at all
  - Few computer scientists even know they exist!
- As we saw, REs/NFAs are usually interpreted as written
  - With some variation in the details of how
- In theory, they could be optimised, like code
  - Several implementations do a little of that
  - Aggressive optimisation would change their behaviour
- Exactly as with expressions in compiled code
  - If you want just the results, that is fine
  - But if you want predictable side-effects, it isn't

The theory in four slides (3)

- A true NFA can be transformed to a unique DFA
  - So-called Deterministic Finite Automaton
  - The details are in many Computer Science books
- Regard this as a form of compilation – it is
- A string of length N is matched in N steps
  - Irrespective of complexity of the original NFA
- Each character is inspected just once per match
  - So can be used to match or reject a stream of input
- Unix *egrep* works like this, and so do some others
- Why don't more utilities use that approach?
  - It is incompatible with most of the above extensions
  - Extended regular expressions are much more powerful

The theory in four slides (4)

- Why were original regular expressions so restrictive?
- Because that form has an efficient (small) DFA
  - I.e. **guaranteed** to be efficient in space and time
- But regular languages are much more general
  - The union of two (alternation) is regular
  - Intersection (i.e. a string matches both A and B)
  - Difference (i.e. a string matches A but not B)
  - Concatenation, repetitions (quantifiers) and so on
- The DFA may be super-exponentially larger than the NFA
  - Which doesn't matter if you are not generating the DFA
  - The time does **not** go up super-exponentially
- You can use a general RE when designing data formats

Theory beyond the above

- It's actually quite easy – for mathematicians!
  Non-mathematicians may find it heavy going
- The following is one book of hundreds

  Introduction to automata theory, languages, and computation by John E. Hopcroft, Rajeev Motwani and Jeffrey D. Ullman, 3rd ed. (2007), Pearson Addison Wesley
- You may also like to look at:

  http://www.cl.cam.ac.uk/teaching/current/...RLFA/reglfa.pdf
- Or, of course, contact the speaker for help

The reserved word problem

- You often want any word except for `fred` or `bert`
  - `alfred`, `frederick`, `alberta` are fine
- This is very common when matching program code
- Original REs don't allow this, but it remains regular
  - So is perfectly good when defining data formats
- Best way to match this in PCRE etc. is something like:

```
((?:fred|bert)\w+|(?!fred|bert)\w+)
((?!(?:fred|bert)\b)\w+)
(\w+(?!\b(?:fred|bert)))
```

- Watch out – `\b` can introduce context dependence
  - In this case, it doesn't, and the above is OK
- The lookahead/behind mustn't go outside the match

The general problem

- In full generality, what we want is:

`A|B`	matches either `A` or `B`	(i.e. alternation)
`A&B`	matches both `A` and `B`	
`A-B`	matches `A` but not `B`	

- I have implemented it, but not seen anyone else do so
  - See the theoretical background for more detail
- You can easily categorise into `A&B` and `A-B`
  - Using the two stage approach to complicated matches
- Step one: find the matches for `A`
- Step two: separate by whether they match `B` or not
- That's how many compilers separate classes of number

The end

Whew!