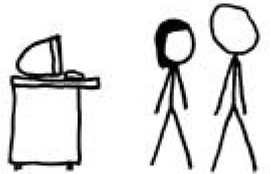
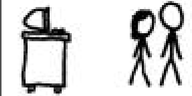
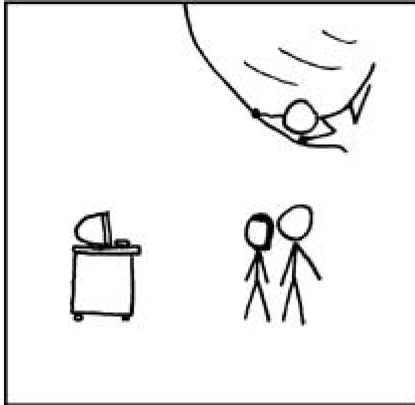
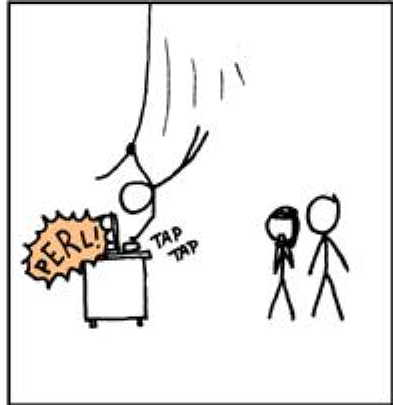
REGular EXpressions

Part II









http://xkcd.com/208

Advanced Computer Methods for Biologists, May 27, 2008
Michael Wrzaczek, michael.wrzaczek@helsinki.fi

Regex Engine Types

There are two fundamentally different types of regex engines: DFA (Deterministic Finite Automation) and NFA (Nondeterministic Finite Automation).

Different programmes and languages use different regex engines.

DFA: awk (most versions), egrep (most versions), MySQL

NFA: GNU Emacs, Java, grep (most versions), less, more, .NET, PCRE (Perl compatible regular expressions) library, Perl, PHP, Python, Ruby, sed, Vim

[POSIX NFA: mawk, GNU Emacs (when requested)]
[Hybrid NFA/DFA: Gnu awk, GNU grep/egrep, TCL]

How Does Pattern Matching Work? (NFA and DFA)

Both regex engines follow 2 rules:

1. The match that begins earliest (leftmost) wins.

2. The standard quantifiers (*, +, ? And {m,n} are greedy.

1. Earliest Match Wins Rule

This rule says, that any match that begins earlier in the string is always preferred over any plausible match that begins later.

The match is first attempted at the very beginning of the string to be searched (just BEFORE the first character). "attempted" means that every permutation of the entire (perhaps complex) regex is tested starting right at that spot. If all possibilities are exhausted and a match is not found, the complete expression is re-tried starting from just before the second character. This full retry occurs at each position in the string until a match is found. No match is reported only after the full retry has been attempted at each position all the way to the end of the string (after the last character).

1. Earliest Match Wins Rule

Trying ORA against FLORAL, the first attempt fails (since ORA can't match FLO). The second attempt also fails (ORA does not match LOR either). The attempt starting at the third position however matches, so the engine stops and reports the match. FLORAL.

1. Why Is This Rule Important?

If you don’t know this rule, some search results might surprise you:

The dragging belly indicates your cat is too fat.

Is you search for ’cat’, the match will be in indicates, not at the word cat, because indicates appears earlier in the string. This is not important in cases like grep, where you just test for the presence of a string, but if you search AND replace the distinction becomes paramount.

Where will this match in the example above:

fat|cat|belly|your

2. The Standard Quantifiers Are Greedy

Greedy means, that the quantifiers will match as many characters as possible. They will settle for something elss than the maximum if they have to, but the always attempt to match as many times as then can up to the absolute maximum allowed.

The only time they settle for anything less than their allowed maximum is when matching too much ends up causing some later part of the regex to fail.

Example:
\b\w+s\b matches words ending with an 's', such as 'regexes'.

The \w+ happily matches the whole word, but if it did, there would be nothing for the s to match. For the match to succeed, \w+ must settle for 'regex' in order for s\b to be able to match.

2. Greedy Quantifiers: First Come, First Served

What is being captured by the parentheses in this example:

String: 'Copyright 2003.'

Regex: `^.*([0-9]+)`

WHY???

NFA vs. DFA

- NFA matches a regex against a given text!

- DFA matches a given text position for position against the regex to be searched!

NFA: Regex-directed

NFA matches a regex against a given text!

Consider a regex to(nite|knight|night) against the text tonight.

Starting with the t, the regular expression is examined one component at a time and the current text is checked to see whether it is matched by the current component of the regex. If it does, the next commponent is checked, and so on, until ALL components have matched, indicating that an overall match has been achieved.

In the example, t is matched first. After the successful match, o is tried and once that matches, the engine moves on to the next expression which is nite or knight or night. Now the engine tries each in turn. The successful match is the last in the example, but the engine won't know before it has all three tried out.

DFA: Text-directed

DFA matches a given text position for position against the regex to be searched!

Consider a regex to(nite|knight|night) against the text tonight.

The DFA engine, while scanning the string, keeps track of all matches "currently in the works". While matching for t and o is similar to the NFA engine, the three alternatives nite, knight and night are evaluate differently. For every option, the first character is evaluated and the list of possible matches is updated. After the engine has reached ton, knight is discarded as impossible while nite and night are kept as possible matches. This continues until nite is ruled out and night is checked until the end to verify the match.

This concludes that generally the DFA engine is faster than the NFA engine

Download files from the server

```
scp -P222 wrzaczek@honkytonk.linux-addict.com:/home/wrzaczek/patternmatch.pl ~/

scp -P222 wrzaczek@honkytonk.linux-addict.com:/home/wrzaczek/dna.fa ~/

scp -P222 wrzaczek@honkytonk.linux-addict.com:/home/wrzaczek/protein.fa ~/

scp -P222 wrzaczek@honkytonk.linux-addict.com:/home/wrzaczek/mailinglist ~/
```

Perl regex-checker

(Detailed discussion of the programme will follow in the Perl lectures!)

```
#!/usr/bin/perl
use strict;
use warnings;

my $file;
my @data;
my $count =1 ;
chomp ($file = <@ARGV>);
open FILE, $file or die "cannot open $file: $!\n";
@data = <FILE>;
for (@data) {
     if (m/REGEX/MODIFIERS) {
          printf "This is a match (Line %02d,        # wrap the line here
                    File $file): $` --- $& --- $'", $count;
          $count++;
     } else {
          $count++;
          next;
     }
}
```

The building blocks: Character Classes (Perl)

Character classes are groups of characters, eg in Perl:

[a-z] uses all characters from a to z while [^a-z] excludes those

[0-9] uses all digits while [^0-9] excludes them, can also be written as \d or \D, respectively

\w identifies a part of word character, usually the same as [a-z0-9_], the exact coverage depends on the system used, but usually all alphanumericals

\W equals [^a-z0-9_] or [^\w]

\s identifies whitespace characters, often the same as [\f\b\r\t\v]

\S non-whitespace characters, the same as [^\s]

. stands for every character except newline.

Anchors

^ anchors a character to the beginning of a string

$ anchors a character to the end of a string

\b anchors a character to a word-boundary (*\B* does the opposite)

In Perl, lookahead and lookbehind offer more advanced possibilities to search for strings followed or preceeded by particular strings.

Lookahead:

Jeff(?=rey) matches Jeff only, if it is part of Jeffrey
(the same as *(?=Jeffrey)Jeff*)
Negative lookahead: *(?!...)*

Lookbehind:

(?<=Jeff)rey matches rey only, if it is part of Jeffrey.
Negative lookbehind: *(?<!...)*

What does *s/(?<=\bJeff)(?=s\b)/''/* do?

Quantifiers

With quantifiers we are able to specify how many instances of
A certain character or character class we want to match.

Quantifiers can be separated into greedy and non-greedy.
Greedy quantifiers will match everything they can while non-greedy ones will only match until a given criterium is matched for the first time.

Greedy quantifiers:

?	Matches one or none (”one optional”)
***	Matches none or unlimited (”any amount ok”)
+	Matches one or unlimited (”at least one”)

{n}	Matches n instances
{m,n}	Matches at least m but at most n instances, matches the maximum possible

Quantifiers

Non-greedy/Lazy quantifiers

These quantifiers only match the minimum required to achieve a successful match.

?? **Matches one or none ("one optional")**
****?*** **Matches none or unlimited ("any amount ok")**
+? **Matches one or unlimited ("at least one")**

{m,n}? **Matches at least m but at most n instances, matches the minimum required**

Quantifiers

Possessive quantifiers:

They work much the same like greedy quantifiers but they NEVER give up a successful match.

?+ Matches one or none ("one optional")
**+* Matches none or unlimited ("any amount ok")
++ Matches one or unlimited ("at least one")

{m,n}+ Matches at least m but at most n instances

They are currently only supported by Java and PHP but mentioned here for completeness. In Perl there are other ways to mimick the behaviour of possessive quantifiers.

Grouping

Within a regex, characters can be grouped using parentheses.

(expression)

Parentheses can be grouping/capturing or grouping/non-capturing.

(expression) is grouping AND capturing

(?:expression) is grouping but non-capturing

What does grouping mean? Regex components are grouped for alternation (an OR statement, see below) and quantification.

(expression){1,3}

What is capturing? One of the most common uses of parentheses is to "pluck" data from a string. The text matched by a parenthesized subexpression is made available after the match. Eg in Perl as the special variables $1, $2, etc…

Atomic grouping

Atomic grouping is indicated in Perl using:

(?>...)

Perl saves all states it goes through during matching. In the case of atomic grouping, matching proceeds normally within the group, but after a successful match, all saved states from within the group are discarded.

This is generally a feature to make matching more efficient and faster. If you are sure, that you do not need saved states from a group, you can use atomic grouping.

In some cases however, it can change the results of your match.

What does this do: *(?>.*?)*

Answer: this is a quite complex way of achieving nothing ☺
Try to explain why (hint: what kind of quantifier is used?)

Alternation

The pipe character | allows to use alternatives for matches.

(expression1|expression2)

```
m/at(?:1|2)g\d{5}/i
```

What will this match?

```
At1g09970
at2G34520
AT1g38750.1

At4g23160
At3g33500.2
```

The syntax of a Perl regex

In Perl, a regex starts with m (for match; the m can be omitted in some cases). The actual regex is delimited by 2 forward slashes.

m/regex/

However, Perl allows you to pick your own pair of delimiters for your expression:

```
m!...!
m,...,
m{...}          s{...}{...}          s{...}!...!
m<...>          s<...><...>
m[...]          s[...][...]
m(...)          s(...)(...)
s|...|...|
qr#...#
```

m identifies a regex for matching

s uses regexes for substitutions

qr accepts a regex just like m and s but allows you to save it as a variable

Modifiers for Perl regexes

After the regular expression in Perl, we can use so-called modifiers.

/i	case insensitive
/s	Lets . Match also newline (\n)
/m	Let the anchors ^ and $ match next to embedded newlines (\n)
/x	Ignore (most) whitespace characters and allow comments within the patterns
/o	Compile the pattern only once
/g	Globally find all matches (matching as many times as possible within the string). In a list context, m/.../g returns a list of all matches found.
/gc	Allow continued search after a failed /g match

About the special characters

You have seen, that some characters are associated with special functions either als character class, quantifiers or delimiters.

What if you want to literally match one of those characters?

You have to ESCAPE them! Eg:

- `.` Will match any character (except newline \n in Perl)
- `\.` Will match .
- `\` Indicates that the following letter serves as a special character (modifier, character class, etc)
- `\\` Will match an actual backslash \.

The x modifier in Perl can be very useful

Breaking down a regex in several lines with whitespace in between can be very helpful in writing clear and understandable regular expressions. Consider a regex to search for a hostname:

```
my $hostnameregex = qr/[-a-z0-9]+(?:\.[-a-z0-9]+)*\.(?:com|org|net|fi)/i;
```

The x modifier in Perl can be very useful

```
my $hostnameregex = qr/[-a-z0-9]+(?:\.[-a-z0-9]+)*\.(?:com|org|net|fi)/i;
```

This can be better written as:

```
    my $hostnameregex = qr{
        [-a-z0-9]+                # the first part at least once
        (?:\.[-a-z0-9]+)*         # the second part not or unlimited
        \.(?:com|org|net|fi)      # the ending exactly once
    }ix;
```

Now we combine this for a full URL:

```
my $httpurl = qr{
    http:// $hostnameregex \b              # the hostname
    (?:
        / [-a-z0-9_:\@&?=+,.!/~*'%\$]*     # optional path
        (?<![.,?!])                        # not allowed at the end
    )?
}ix;
```

Note: within a character class, special symbols do not need to be escaped. However, in the above example the @ and $ are special. Perl uses @ and $ to identify variables and interprets them in that way even within character classes. Therefore they need to be escaped!

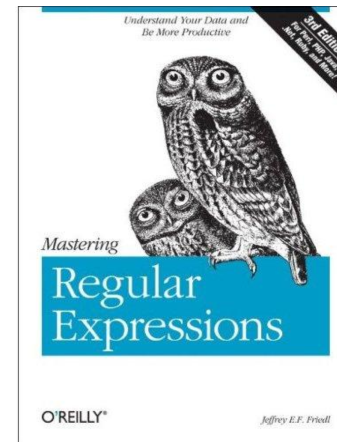
Where to go from here?

Regular expressions are a quite complicated topic, we barely scratched the surface here. We did not address different types of regex engines and we also did not touch the topic of the performance and efficiency of regular expressions.

Suggested further reading:

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

THE regex bible! Covers almost every aspect of regular expressions.



Regular Expressions Pocket Reference
Tony Stubblebine, O'Reilly

A quick and good reference to regexes in most Unix tools and scripting languages. Requires however understanding of regular expressions.