Lectures 20 and 21: Regular Expressions

36-350, Fall 2012

7 and 12 November 2012

With basic string-manipulation functions, we saw how to do things like split up entries in a data file which are separated by commas

```
strsplit(text,split=",")
```

or by single spaces

```
strsplit(text,split=" ")
```

or even a comma followed by a space

```
strsplit(text,split=", ")
```

But we don't know how to deal with situations like splitting on a comma, *optionally* followed by some number of spaces.

Not only is it annoying to have such a simple thing defeat us, it's an instance of a much broader class of problems. If we're trying to extract data from webpages, we may want to get rid of all the formatting instructions buried in the source of the webpage. We might want to extract all the personal names from a document which are preceded by titles (such as Mr., Ms., Miss, Dr.), without knowing what those names are, or how long they are. And so forth.

What all these examples have in common is that we are looking, not for particular strings, but for strings which fit certain patterns. This is something the computer can do for us, if we can provide it with an algorithm to decide whether or not a string any given conforms to the pattern.

There are such algorithms for certain kinds of text patterns, those which are described in a formalism called **regular expressions**. We will *not* go into those algorithms; instead, we will focus on crafting and using regular expressions, and say just a little bit about their limitations[1].

1 The Rules of Regular Expressions

Every regular expression is a sequence of symbols, which specifies a set of text strings that follow some pattern that — **match** the regular expression. A valid regular expression must conform to certain rules of grammar; it gets interpreted

[1]See "Further Reading" at the end for more.

by the computer as rules for matching certain strings, but not others. Let us build up the syntax for regexp — the rules for what makes a valid expression — side by side with the semantics, how the expression is interpreted.

1. Every string is a valid regular expression. It matches instances of that string. So `fly` is a regular expression which gets matched to the end of "fruitfly", "horsefly", "why walk when you can fly", and "watch the fur fly". It does not match any part of "time flies like an arrow; fruit flies like a banana; a banana flies poorly".

2. The **concatenation** of two regular expressions is also another valid regular expression. Strings match the concatenation if and only if they match each part, in order, without gaps. `fly` is really the concatenation of `f`, `l`, `y`.

3. We indicate the logical-OR of two regular expressions with the vertical bar `|`. Thus `fly|flies` has matches in all the above strings. OR-ing can be repeated: `fly|flies|Fly|Flies`.

4. We can precede any character with a special meaning in a regular expression, such as `|`, with the **escape** character, `\`. This keeps the character from being interpreted in its special sense, just in its literal meaning. Thus `X\|y` matches part of "Pr(X|y)", while `X|y` matches three times in "syzygy", and twice in "Xeroxy".

5. We indicate a set of characters, any of which will work, with square braces, `[ ]`. Thus `M[rs]` matches "Mr" or "Ms". (To match a literal `]`, put it first in the list: `[])}>]` matches all right-hand delimiters.)

   (a) We can indicate a range of characters inside braces with a dash: `[A-Z]` indicates any upper-case letter, `[4-7]` the numerals 4, 5, 6, 7, etc. To match a literal hyphen, put it first or last in the list.

   (b) Some sets of characters are common enough that there are pre-defined classes for them; see `help(regexp)` for a complete list. The ones which are common enough to be worth mentioning here are:

   - `[:lower:]` and `[:upper:]`, the upper and lower case letters, and `[:alpha:]`, all letters of the alphabet;
   - `[:digit:]`, the numerals 0–9
   - `[:alnum:]`, the alphanumeric characters (union of `[:alpha:]` and `[:digit:]`)
   - `[:punct:]`, all the punctuation marks
   - `[:space:]`, all the whitespace characters (also `\s`)
   - `\w`, word characters, i.e., the alphanumeric characters and the underscore, `[[:alnum:]_]` ; `\W` is every character not in the `\w` class

   (c) We can indicate “every character *except* those in this range” with an initial caret or hat, `^`, inside the braces. Thus `[^aeiou]` matches every character except a lower-case vowel. (To include a literal caret in the list of matches, but it anywhere except the first position.)
   (d) The special character `.` stands for the class of *all* characters. Thus `M..s` matches “Miss”, “Mass”, “Mess”, “Mits”, “Mats”, “M11s”, “M s”, etc. Note that the dot does not have to be enclosed in brackets. Also note that when we have two (or more) dots in a row, the matching characters do not have to be the same.

6. Any valid sub-expression can be repeated various numbers of times, by means of **quantifiers**.
   (a) `+` means, match if the expression is repeated one or more times.
   (b) `*` means, match if the expression is repeated 0 or more times. (That is, the expression is optional, but if it appears at all, it can appear any number of times.)
   (c) `?` means, match if the expression appears 0 or 1 times. (That is, the expression is optional but cannot be repeated if it does show up.)
   (d) `{n}` means, match if the expression is repeated exactly n times.
   (e) `{n,}` means, match if the expression is repeated n or more times.
   (f) `{n,m}` means, match if the expression is repeated between n and m times (inclusive).

   For instance, `M[rs][rs]?\.?` matches “Mr.”, “Mr”, “Ms”, “Ms.”, “Mrs”, “Mrs.” and “Mrs”, but not “Miss”. The regexp `M[rs][rs]?\.?[ ]+[A-Z]` matches `Mr. A`, `Mrs.  B` (note the two spaces), but not `Ms.C`.

7. By default, all quantifiers are “greedy”, and match as many repetitions as possible. Any quantifier followed by `?` will instead match the smallest number of repetitions: If we try to match `\[.+\]` in `[i][j]`, it will match the whole expression, whereas `\[.+?\]` will just match `[i]`.

8. By default, quantifiers apply to the last character (or character range) before they appear. Any valid-sub-expression however can be enclosed in parentheses, `( )`, for grouping. Thus `H(TT)+` matches a single H, followed by an *even* number of T’s. `(HH|TT)+` will match any pattern of H’s and T’s, provided that each comes in blocks of even length — thus “HHTTHHHHTTHH”, but not “HHTHHHHTTHH”.

9. `$` means that a pattern can only match at the end of a line — e.g., `[a-z,]$` finds lines that end in a lower-case letter or a comma — while `^` (outside of braces!) similarly anchors a pattern at the beginning; `^[^A-Z]` matches strings that begin with something other than a capital letter. `\<` and `\>` anchor to the beginning and end of words (respectively), `\b` matches either the beginning or ending of a word (think “`\b` for boundary”), and `\B` matches anywhere except the beginning and ending of words.

10. Finally, there are **back-references**, indicated by `\1`, `\2`, ...`\9`. These refer to whatever matched has already the first, second, ...ninth sub-expression in parentheses. Thus `[HT]+` will match any string of H's and T's, but `([HT])+\1` will only match strings of H's and T's where the second have exactly repeats the first. The content of the back-reference is sometimes called a "capture", or "the captured string".

1.0.1 Variables Containing Regular Expressions

You will have noticed that a regular expression is itself just a string. It can therefore be stored in a `character`-type variable, built up using the usual string-manipulation commands (e.g., `paste`), or itself subjected to regular expression matching and modification. Some of this will be illustrated below.

2 Commands Using Regular Expressions

2.1 `strsplit`

We have already seen one command that uses a regular expression, `strsplit`.

Last time, we tried splitting `al2` into words by splitting on spaces:

```
al2 <- readLines("http://www.stat.cmu.edu/~cshalizi/statcomp/lectures/19/al2.txt")
al2 <- paste(al2, collapse=" ")
al2.words <- strsplit(al2, split=" ")[[1]]
```

This produced some strange results, since punctuation marks got treated as parts of words. We also got some weirdness from situations where there were multiple spaces in a row.

```
al2.words <- strsplit(al2, split="(\\s|[[:punct:]])+")[[1]]
```

This looks for blocks containing only whitespace and/or punctuation. Unfortunately this splits apart possessive, indicated by apostrophes; "men's" splits into "men" "s". (If Lincoln used contractions, like "don't", they would also be split.) We want to express *either* any number of white spaces, *or* at least one punctuation mark followed by at least one space.

```
al2.words <- strsplit(al2, split="\\s+|([[:punct:]]+[[:space:]]+)")[[1]]
```

You can check that now, e.g., "men's" is handled properly.

Notice, in all of this, that the regular expression gets enclosed in quotation marks — we find it in to R as just another string. Because we want that string to contain backslashes, we need to write double-backslashes. (Why?)

2.1.1 grep and grepl

grep[2] scans a character vector for all occurrences of a regular expression. It returns either the indices of the vector with matches, or the actual matching strings.

To illustrate, consider the file ANSS.csv.html on the class website, which was generated from http://www.quake.geo.berkeley.edu/anss/catalog-search.html; it contains a catalog of every earthquake of at least magnitude 6 on the Richter scale, from 1 January 2002 to 1 January 2012[3]. Here are its first dozen lines:

```
<HTML><HEAD><TITLE>NCEDC_Search_Results</TITLE></HEAD><BODY>Your search parameters are:<ul>
<li>catalog=ANSS
<li>start_time=2002/01/01,00:00:00
<li>end_time=2012/01/01,00:00:00
<li>minimum_magnitude=6.0
<li>maximum_magnitude=10
<li>event_type=E
</ul>
<PRE>
DateTime,Latitude,Longitude,Depth,Magnitude,MagType,NbStations,Gap,Distance,RMS,Source,EventID
2002/01/01 10:39:06.82,-55.2140,-129.0000,10.00,6.00,Mw,78,,,1.07,NEI,2002010140
```

The first lines are HTML formatting directions, and search parameters. The actual data only begins in line 12. Every line of the *data* begins with a date in the format YYYY/MM/DD. To extract which lines those are, and then the actual pattern-matching lines[4]:

```
> anss <- readLines("ANSS.csv.html",warn=FALSE)
> head(grep(x=anss,pattern="^[0-9]{4}/[0-9]{2}/[0-9]{2}"))
[1] 11 12 13 14 15 16
> head(grep(x=anss,pattern="^[0-9]{4}/[0-9]{2}/[0-9]{2}",value=TRUE))
[1] "2002/01/01 10:39:06.82,-55.2140,-129.0000,10.00,6.00,Mw,78,,,1.07,NEI,2002010140"
[2] "2002/01/01 11:29:22.73,6.3030,125.6500,138.10,6.30,Mw,236,,,0.90,NEI,2002010140"
[3] "2002/01/02 14:50:33.49,-17.9830,178.7440,665.80,6.20,Mw,215,,,1.08,NEI,2002010240"
[4] "2002/01/02 17:22:48.76,-17.6000,167.8560,21.00,7.20,Mw,427,,,0.90,NEI,2002010240"
[5] "2002/01/03 07:05:27.67,36.0880,70.6870,129.30,6.20,Mw,431,,,0.87,NEI,2002010340"
[6] "2002/01/03 10:17:36.30,-17.6640,168.0040,10.00,6.60,Mw,386,,,1.14,NEI,2002010340"
```

Since it's irritating to have to keep typing the regular expression over and over, let's store it in a variable:

```
> initial_date <- "^[0-9]{4}/[0-9]{2}/[0-9]{2}"
> all.equal(grep(x=anss,pattern="^[0-9]{4}/[0-9]{2}/[0-9]{2}"),
+   grep(x=anss,pattern=initial_date))
[1] TRUE
```

[2] *G*lobal *r*egular *e*xpression *p*arser (or *p*rinter); a name which has been fossilized in Unix and related systems since 1973.

[3] The website is an interface to a much larger database, extending further back in time, and capable of filtering geographically and by magnitude.

[4] The warn=FALSE option turns to readLines turns off a superfluous warning here about the file not ending with a special end-of-file, or even end-of-line, character.

Note that if we want stuff which does *not* match our pattern, we have an `invert` option:

```
> grep(x=anss,pattern=initial_date,invert=TRUE,value=TRUE)
 [1] "<HTML><HEAD><TITLE>NCEDC_Search_Results</TITLE></HEAD><BODY>Your search parameters are:<ul>"
 [2] "<li>catalog=ANSS"
 [3] "<li>start_time=2002/01/01,00:00:00"
 [4] "<li>end_time=2012/01/01,00:00:00"
 [5] "<li>minimum_magnitude=6.0"
 [6] "<li>maximum_magnitude=10"
 [7] "<li>event_type=E"
 [8] "</ul>"
 [9] "<PRE>"
[10] "DateTime,Latitude,Longitude,Depth,Magnitude,MagType,NbStations,Gap,Distance,RMS,Source,EventID"
[11] "</PRE>"
[12] "</BODY></HTML>"
```

The related command `grepl` returns a Boolean vector, indicating whether or not there were matches, for each element of the character vector:

```
> tail(grepl(x=anss,pattern=initial_date))
[1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE
```

This information could be recovered from the `value=FALSE` version of `grep`, of course, but `grepl` is much faster, when that's what's needed.

2.1.2 `regexpr`,`gregexpr`, `regmatches`

These all return information about where regular expressions are matched in a string. `regexpr` returns the location of the first match (or -1 if there isn't any match), along with attributes like the length of the match. `gregexpr` works similarly, but gives information about all matching locations, in a list. (Here, and in many regular-expression functions, the initial `g` stands for "global".) `regexpr` and `gregexpr` return `-1` for all elements of the vector they'r applied to where there is no match.

`regmatches` takes strings and the output of `regexpr` or `gregexpr`, and returns the actual matching strings.

For instance, this extracts all the (latitude, longitude) pairs of coordinates from the earthquake data:

```
> one_geo_coord <- paste("-?[0-9]+\\.[0-9]{4}")  # \\ to get a literal \
> pair_geo_coords <- paste(rep(one_geo_coord,2),collapse=",")
> have_coords <- grepl(x=anss,pattern=pair_geo_coords)
> coord.matches <- gregexpr(pattern=pair_geo_coords,text=anss[have_coords])
> coords <- regmatches(x=anss[have_coords],m=coord.matches)
> head(coords)
[[1]]
[1] "-55.2140,-129.0000"
[[2]]
[1] "6.3030,125.6500"
```

```
[[3]]
[1] "-17.9830,178.7440"
[[4]]
[1] "-17.6000,167.8560"
[[5]]
[1] "36.0880,70.6870"
[[6]]
[1] "-17.6640,168.0040"
```

Notice that we give `regmatches` a vector, and it gives us back a list — strictly, a list of character vectors of matches.

We can of course do more processing of the matches:

```
> coords <- do.call(c,coords)  # De-list-ify to vector
> coord.pairs <- strsplit(coords,",")  # Break apart latitude and longitude
> coord.df <- do.call(rbind, coord.pairs) # De-list-ify to array
> coord.df <- apply(coord.df,2,as.numeric) # Character to numeric
> coord.df <- as.data.frame(coord.df)
> colnames(coord.df) <- c("Latitude","Longitude")
> head(coord.df)
  Latitude Longitude
1  -55.214  -129.000
2    6.303   125.650
3  -17.983   178.744
4  -17.600   167.856
5   36.088    70.687
6  -17.664   168.004
```

leading to things like Figure 1.

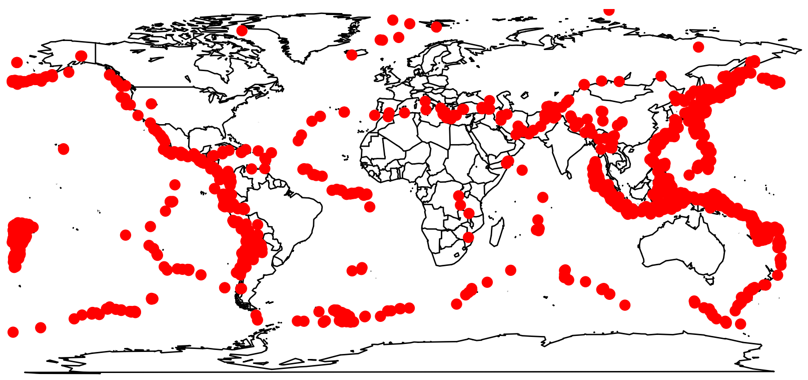
Having to go indirectly through a data structure of matching locations, and then to the actual matching strings, may seem needlessly indirect, but it allows us to do things like easily count the number of matches, or how long they are, without having to actually extract them, and we can even see what text in one file corresponds to matching locations in another file.

2.2 `regexec`

Often, we only care about certain parts of the string which matches a regular expression.

We might, for instance, want to find the names of all the people referred to in a document as "Mr. X" or "Ms. Y". The initial title matters, it lets us pick out the names, but we only want the names, not the names-and-titles. Faced with this text

```
Ms. Alice B. Tolkas (lately of Paris) will be discussing brownie recipes with
Mr. Harold Lee and Mr. Kumar Patel (of New Jersey), in the J. R. "Bob" Dobbs
Memorial Auditorium, at 3 p.m.  The Rev. Mr. Robert Burton, of Brasenose
College, will moderate.
```



```
library(maps)
map("world")
points(coord.df$Longitude,coord.df$Latitude,pch=19,col="red")
```

Figure 1: Geographic distribution of earthquakes, extracted from a data file using regular expression matching.

we would want to get "Alice B. Tolkas", "Harold Lee", "Kumar Patel", and "Robert Burton". How do we do this?

We can get much of the way with what we already know:

```
> alice_bob <- paste(readLines("alice-bob.txt"),collapse=" ")
> honorific_names <- "(Mr|Ms)\\. ([A-Z]([a-z]+|\\.)[ ]+)*[A-Z][a-z]+"
> regmatches(alice_bob,gregexpr(honorific_names,alice_bob))
[[1]]
[1] "Ms. Alice B. Tolkas" "Mr. Harold Lee"      "Mr. Kumar Patel"
[4] "Mr. Robert Burton"
```

(Unpack the regular expression: what it says expresses is that, after the title, the period, and a space, a name consists of an alternation of words only whose initial letter is capitalized, or capital letters with periods, ending in a word only whose initial letter is capitalized. This will work for all the names given above, and, manifestly, for many common Anglicized names.[5])

One use of parentheses in a regular expression is to mark a sub-pattern for latter use, a **capture group**. The command `regexec` works like `regexpr`, but returns information about all the capture groups. So let's enclose the name part in parentheses, and see what happens:

```
> honorific_names <- "(Mr|Ms)\\. (([A-Z]([a-z]+|\\.)[ ]+)*[A-Z][a-z]+)"
> regmatches(alice_bob,regexec(honorific_names,alice_bob))
[[1]]
[1] "Ms. Alice B. Tolkas" "Ms"                  "Alice B. Tolkas"
[4] "B. "                 "."
```

We're getting the over-all matching string, and the matches for each parenthesized sub-string — but *only* for the very first match. This is like what we'd see if we ran `regexpr` instead of `gregexpr`. The usual work-around is to do things in two stages:

```
> (name_matches <- regmatches(alice_bob,gregexpr(honorific_names,alice_bob))[[1]])
[1] "Ms. Alice B. Tolkas" "Mr. Harold Lee"      "Mr. Kumar Patel"
[4] "Mr. Robert Burton"
> (name_parts <- regmatches(name_matches,regexec(honorific_names,name_matches)))
[[1]]
[1] "Ms. Alice B. Tolkas" "Ms"                  "Alice B. Tolkas"
[4] "B. "                 "."

[[2]]
[1] "Mr. Harold Lee" "Mr"             "Harold Lee"     "Harold "
[5] "arold"
```

[5]It will fail if it encounters a name written as "J B S Haldane" (as opposed to "J. B. S. Haldane"), or "D'Arcy Wentworth Thompson", or "John von Neumann", or "Th. de Quincey", or "Esteban Maturin y Domanova", or "'Abd-ar-Rahmán Abû Zayd ibn Muhmammad ibn Muhammad ibn Khaldûn". It is surprisingly easy for text-analysis projects to run aground on parochial assumptions like the one about the form of names embedded in the regular expression. It is also surprisingly hard to make this point to programmers without seeming smugly righteous.

```
[[3]]
[1] "Mr. Kumar Patel" "Mr"              "Kumar Patel"
[4] "Kumar "          "umar"

[[4]]
[1] "Mr. Robert Burton" "Mr"                "Robert Burton"
[4] "Robert "           "obert"
> (names <- sapply(name_parts,function(x) { x[3] }))
[1] "Alice B. Tolkas" "Harold Lee"      "Kumar Patel"
[4] "Robert Burton"
```

Of course, if we also wanted the titles, we could get them too:

```
> (names_and_titles <- t(sapply(name_parts,function(x) { c(x[3],x[2]) })))
     [,1]              [,2]
[1,] "Alice B. Tolkas" "Ms"
[2,] "Harold Lee"      "Mr"
[3,] "Kumar Patel"     "Mr"
[4,] "Robert Burton"   "Mr"
```

2.3 Replacements

`regmatches` can be used to replace matching parts of a string (or strings), just like `substr`.

```
> baking <- c("Bake your brownie at 350F for 7--10 minutes")
> regmatches(baking,gregexpr(pattern="[0-9]+F",text=baking)) <- "some temperature"
> baking
[1] "Bake your brownie at some temperature for 7--10 minutes"
```

More complex substitutions are of course possible. Note that one needs to provide a list for multiple replacements to work properly.

```
> baking <- c("Bake your brownie at 350F for 7--10 minutes, let cool to 100F before cutting")
> F_temps <- regmatches(baking,gregexpr("([0-9]+)F",baking))[[1]]
> temps_in_F <- regmatches(F_temps,regexec("([0-9]+)F",F_temps))
> temps_in_F <- as.numeric(sapply(temps_in_F,function(x){x[2]}))
> temps_in_C <- signif((temps_in_F-32)*100/180,2)
> C_temps <- paste(temps_in_C,"C",sep="")
> regmatches(baking,gregexpr("([0-9]+)F",baking)) <- list(C_temps)
> baking
[1] "Bake your brownie at 180C for 7--10 minutes, let cool to 38C before cutting"
```

The functions `sub` and `gsub` work like `regexpr` and `gregexpr`, but also take a `replace` argument, which gets substituted for either the first or for all matches. This can include back-references, numbered `\1` through `\9`, to capture groups in the match. Unlike assigning to `regmatches`, it does not change the string it's applied to, it returns a new string with the substitution:

To illustrate, this removes everyone's honorifics:

```
> gsub(pattern=honorific_names,replacement="\\2",x=alice_bob)
[1] "Alice B. Tolkas (lately of Paris) will be discussing brownie recipes with
 Harold Lee and Kumar Patel (of New Jersey), in the J. R. \"Bob\" Dobbs
 Memorial Auditorium, at 3 p.m.  The Rev. Robert Burton, of Brasenose College,
 will moderate."
```

3 Challenge

`ss.html` contains the text of an actual webpage, "encountered in the wild"[6]. Extract *all* the text, and *only* the text, discarding HTML formatting and hyperlinks (but *not* the text the link is anchored to).

4 Further Reading

J. E. F. Friedl's *Mastering Regular Expressions* is a very useful resource for more advanced work. It focuses on the language Perl, which is a valuable resource for intensive text processing, but most of the ideas apply very readily to other tools which use regular expressions. (Perl's syntax for regular expressions differ very slightly from the "POSIX" standard which is the default in R, but there are also options for using Perl's syntax in R.)

`http://regexlib.com/` is a useful website for regular expressions for various purposes. These are user-contributed, so there's no guarantee of correctness, but they are free, and many of the contributors know what they're doing.

The oldest treatment of regular expressions is [5], which can be seen as a follow-up to earlier attempts [9] to understand the computational power of simple nervous systems, *how* electrical impulses in the brain could *possibly* be doing logic.

A regular expression is an example of a **formal grammar**, which is a set of rules specifying which strings are part of a **language**, and which are not. There is actually a hierarchy of increasingly powerful and expressive types of formal grammars, first investigated by Noam Chomsky in the 1950s [2, 3][7]. Regular expressions are at the very bottom of this hierarchy. Computationally, they correspond to patterns which require only a *fixed* (and finite) number of bits of memory to decide — a computer with only one bit of memory can decide whether an arbitrarily long string of heads and tails matches `(H|(TT))+` (by tracking whether it has seen an odd or an even number of tails since the last head). Higher levels of the hierarchy require more memory; for instance, no finite memory can decide whether *every* string of heads and tails is a palindrome. Regular expressions correspond to **finite-state machines**, or **finite automata**. There are formalisms like those of regular expressions for describing higher levels of the

[6]`http://simplystatistics.org/post/35187901781/nate-silver-does-it-again-will-pundits-finally-accept`

[7]Chomsky, in turn, was drawing on an earlier tradition of work on formal languages and mathematical logic by the Logical Positivists, in the 1930s and 1940s.

hierarchy, but the higher one goes in the hierarchy, the harder it is, computationally, to decide whether a given string confirms to the grammar. Good introductions to this topic, which is one of the foundations of computer science, include [10], [4], [6] and `http://www.santafe.edu/~moore/automata-notes.pdf`.

Probabilistic generalizations or extensions of regular expressions correspond to various sorts of Markov models, where the state is not directly observable; these are *extremely important* for natural-language processing algorithms. [1], while older than some of you, is still very sound and readable. [7] is more comprehensive.

WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP

Figure 2: There are many uses for regular expressions; here, they are shown fueling a programmer's fantasy life. (From `http://xkcd.com/208/`.)

References

[1] Charniak, Eugene (1993). *Statistical Language Learning*. Cambridge, Massachusetts: MIT Press.

[2] Chomsky, Noam (1956). “Three Models for the Description of Language.” *IRE Transactions on Information Theory*, **2**: 113–124.

[3] — (1957). *Syntactic Structures*. The Hauge: Mouton.

[4] Hopcroft, John E. and Jeffrey D. Ullman (1979). *Introduction to Automata Theory, Languages, and Computation*. Reading: Addison-Wesley. 2nd edition of *Formal Languages and Their Relation to Automata*, 1969.

[5] Kleene, S. C. (1956). “Representation of Events in Nerve Nets and Finite Automata.” In *Automata Studies* (Claude E. Shannon and John McCarthy, eds.), pp. 3–41. Princeton, New Jersey: Princeton University Press.

[6] Lewis, Harry R. and Christos H. Papadimitriou (1998). *Elements of the Theory of Computation*. Upper Saddle River, New Jersey: Prentice-Hall, 2nd edn.

[7] Manning, Christopher D. and Hinrich Schütze (1999). *Foundations of Statistical Natural Language Processing*. Cambridge, Massachusetts: MIT Press.

[8] McCulloch, Warren S. (1965). *Embodiments of Mind*. Cambridge, Massachusetts: MIT Press.

[9] McCulloch, Warren S. and Walter Pitts (1943). “A logical calculus of the ideas immanent in nervous activity.” *Bulletin of Mathematical Biophysics*, **5**: 115–133. Reprinted in [8, pp. 19–39].

[10] Minsky, Marvin (1967). *Computation: Finite and Infinite Machines*. Englewood Cliffs, New Jersey: Prentice-Hall.