Corpus Linguistics
(L415/L615)
Regular expressions

Markus Dickinson
Department of Linguistics, Indiana University
Fall 2015
Motivating regular expressions

**Regular expressions** help describe complex patterns of words and text

- Find *help to* *V* constructions in POS-tagged text, with words & tags mixed up together
- Retrieve the first verb used in a relative clause
- Find all Indiana email addresses occurring in a long text
Regular expressions: What they are

- A regular expression is a compact description of a set of strings, i.e., a language (in **formal language** theory)
  - They are used to search for occurrences of these strings
- Regular expressions can only describe so-called **regular languages**
  - Some patterns cannot be specified using regular expressions, e.g., finding a string containing an arbitrary number of matching parentheses
Regular expressions: Tools that use them

- A variety of unix tools (grep, sed, . . .), editors (emacs, jEdit, . . .), and programming languages (perl, python, Java, . . .) incorporate regular expressions.
  - We’ll start with grep & then move to perl
- Some of the concordancing tools you’ve seen (e.g., AntConc) allow for regular expression searching.
- Implementations are very efficient so that large text files can be searched quickly

The various tools differ w.r.t. the exact syntax of the regular expressions they allow, but knowledge of one transfers
The syntax of regular expressions (I)

Regular expressions consist of

- strings of literal characters: c, A100, natural language, 30 years!
- disjunction:
  - ordinary disjunction: devoured|ate, famil(y|ies)
  - character classes: [Tt]he, bec[oa]me
  - ranges: [A–Z] (any capital letter)
- negation:
  -[^a] (any symbol but a)
  -[^A–Z0–9] (not an uppercase letter or number)
Specific character classes

Use **aliases** to designate particular recurrent sets of characters

- `\d = [0-9]`: digit
- `\D = [^\d]`: non-digit
- `\w = [a-zA-Z0-9_]`: alphanumeric
- `\W = [^\w]`: non-alphanumeric
- `\s = [\r\t\n\f]`: whitespace character
  - `\r`: space, `\t`: tab, `\n`: newline, `\f`: formfeed
- `\S [^\s]`: non-whitespace
The syntax of regular expressions (II)

▶ counters
  ▶ optionality: ?
    colou?r
  ▶ any number of occurrences: * (Kleene star)
    [0-9]* years
  ▶ at least one occurrence: +
    [0-9]+ dollars

▶ wildcard for any character: .
  beg.n for any character in between beg and n

▶ Parentheses to group items together
  ant(farm)?

▶ Escaped characters to specify characters with special meanings:
  \*, \+, \?, \(, \), \|, \[, \]
The syntax of regular expressions (III)

- Operator precedence, from highest to lowest:
  - parentheses ()
  - counters * + ?
  - character sequences
  - disjunction |

- fire|ing = fire or ing
- fir(e|ing) = fir followed by either e or ing
The syntax of regular expressions (IV)

Anchors: anchor expressions to various parts of the string

▶ ^ = start of line
  ▶ do not confuse with [^ . . . ] used to express negation
▶ $ = end of line
▶ \b = non-word character (i.e., word boundary)
  ▶ word characters are digits, underscores, or letters, i.e.,
    [0-9A-Za-z_]

Instead of writing out specific numbers of occurrences, repetition can be represented between \{ \}

▶ a{4} = 4 a’s
▶ a{1,4} = 1-4 a’s
Some RE practice

- What does `\$[0-9]+(\.[0-9][0-9])` signify?
- Write a RE to capture the times on a digital watch (hours and minutes). Think about:
  - the (im)possible values for the hours
  - the (im)possible values for the minutes
grep is a powerful and efficient program for searching in text files using regular expressions.

- It is standard on Unix, Linux, and Mac OS X, and there also are various ports to Windows (e.g., http://gnuwin32.sourceforge.net/packages/grep.htm, http://www.interlog.com/~tcharron/grep.html or http://www.wingrep.com/).

- The version of grep that supports the full set of operators mentioned above is generally called egrep (for extended grep).
In the following, we assume a text file `f.txt` containing, among others, the strings that we mention as matching.

- **Strings of literal characters:**
  ```bash
egrep 'and' f.txt
  ```
  matches `and`, Ayn Rand, Candy and so on

- **Character classes:**
  ```bash
egrep 'the year [0-9][0-9][0-9][0-9]' f.txt
  ```
  matches the year 1776, the year 1812, the year 2001, and so on
Grep: Examples for using regular expressions (II)

- **disjunction (|):** `egrep 'couch|sofa' f.txt` matches couch or sofa

- **grouping with parentheses:**
  `egrep 'un(interest|excit)ing' f.txt` matches uninteresting or unexciting.

- **Any character (.):**
  `egrep 'o.e' f.txt` matches ore, one, ole
Grep: Examples for using regular expressions (III)

- Kleene star (*):
  ```
  egrep 'a*rgh' f.txt matches argh, aargh, aaargh
  ```

- One or more (+):
  ```
  egrep 'john+y' f.txt matches johny, johnny, ..., but not johy
  ```

- Optionality (?):
  ```
  egrep 'joh?n' f.txt matches jon and john
  ```
Various online web interfaces allow RE or RE-like queries

- To provide efficient searching in large corpora, in these search engines regular expressions over characters are often limited to single tokens (i.e. generally words)

- **BNC:**
  - web form: 
    - http://www.natcorp.ox.ac.uk/using/index.xml?ID=simple
  - regular expressions are enclosed in { }

- **Internet corpora:**
  - http://corpus.leeds.ac.uk/internet.html
  - See notes on query language: 
    http://corpus.leeds.ac.uk/help.html
help/help to

Now on to a question for us to investigate . . .

Subcategorizations of help:

- help to V
- help NP to V
- help V
- help NP V

Some questions:

- Is the choice of to arbitrary?
- What factors influence this selection?
Getting started

▶ Getting a concordancer
  ▶ We know how to use AntConc, but we want to scale up with our own programs
  ▶ And we’ll want to extract frequencies
▶ Getting corpus data: for this to work, you’d want to look at 2 different corpora
  ▶ We’re going to only look at one during class, the Brown corpus, found on miller at:
    ▶ /Volumes/Data/en/brown/
  ▶ You can do comparisons outside of class, assuming (similar) POS tags in the other corpus
Searching for our specific patterns

We can use AntConc or we can use Perl (see help.pl), which is well-maintained for regular expressions

We want to compare help V and help to V
  ▶ and to compare help NP V and help NP to V
  ▶ For these latter cases, we’ll simplify the NP to be a single noun (tag starts with n) or pronoun (p)

The patterns we’ll use (read \s+ as a space):
  ▶ help V: \b(help\w*?/v\w*?\s+\w+/v\w*?)\b
  ▶ help to V:
    \b(help\w*?/v\w*?\s+to/to\s+\w+/v\w*?)\b
  ▶ help NP V [approximated]:
    \b(help\w*?/v\w*?\s+\w+/[np]\w*?\s+\w+/v\w*?)\b
  ▶ help NP to V [approximated]:
    \b(help\w*?/v\w*?\s+\w+/[np]\w*?\s+to/to\s+\w+/v\w*?)\b
Breaking down the regular expression

For now

```
while (m{\b(help\w*?/v\w*?\s+\w+/v\w*?)\b}gi)
```

Perl options

- Perl is matching the current line against this regular expression, within `m{...}`
- The `g` at the end specifies *global* matching, in case there is more than one match
  - This is also why we have a `while` loop
- The `i` at the end makes it case-insensitive
Breaking down the regular expression (2)

```regex
while (m{\b(help\w*?/v\w*?\s+/v\w*?)\b}gi)
```

This is the formal way of saying *help* (or its variants), used as a verb (*v*) and followed by any verb

- `\b` specifies a word boundary
- `\w` specifies only word characters, and `*?` specifies 0 or more of them, i.e., possible word endings
  - FYI (for those “in the know”): `*?` is the non-greedy version of the usually greedy `*`
- `\s` specifies white-space characters, and `+` specifies one or more of them
  - this is more robust than just putting a space
Breaking down the regular expression (3)

This regular expression:
\b(help\w*?/v\w*?\s+\w+/v\w*?))\b

is trying to say this:
help(s|ed|ing|ful|...)/verb <word>/verb

Today’s goal 1: Get the gist of this expression

▶ Today’s goal 2: feel comfortable making little changes to search for what you want

▶ Today’s goal 3 (later): unpacking the RE a bit more
Frequency counts

Let’s get the frequency counts . . .

<table>
<thead>
<tr>
<th>Inf-type</th>
<th>No NP</th>
<th>With NP</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bare</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

22 / 28
Intervening NP

Take a look at the numbers you have for *help (to)* in Brown

- with an intervening NP
- without an intervening NP

Do you notice any trends?
Infinitive marker

A further claim:
- if *to* precedes *help*, then there is less likely to be a *to* afterwards

Let’s alter the Perl regular expressions to find out whether this is true in the Brown data
- What types of patterns do we need to express?
- Can we use particular POS tags to help?
The passive

Yet a further claim:

- The passive (e.g., *were helped*) only occurs with *to*

So, let’s search for:

- *be* verbs (POS starting with *vb*) followed by a *vvn help*
  - with vs. without *to* + following verb

We’ll cover statistical tests in the upcoming weeks, to see whether the observed differences are statistically significant

- Without statistical tests for now, what do you observe in the table?
Revisiting our patterns

We’ve compared help V & help to V

▶ and help NP V & help NP to V

▶ For these latter cases, we simplified the NP to be a single noun (tag starts with n) or pronoun (p)

The patterns we used:

▶ help V: \b(help\w*?/v\w*?\s+\w+/v\w*?)\b

▶ help to V:
  \b(help\w*?/v\w*?\s+to/to\s+\w+/v\w*?)\b

▶ help NP V:
  \b(help\w*?/v\w*?\s+\w+/[np]\w*?\s+\w+/v\w*?)\b

▶ help NP to V:
  \b(help\w*?/v\w*?\s+\w+/[np]\w*?\s+to/to\s+\w+/v\w*?)\b
Breaking down the regular expression

\b(help\w*/v\w*/?\s+\w+/v\w*/?)\b

So, what do we see here?

- Word boundaries before `help` & at the end
- `help` followed by a sequence of 0 or more (*) word characters (\w)
  - This matches `help`, `helps`, `helpful`, etc.
  - We’ll talk about `*`? momentarily
- `/v\w*/?`: this matches a string starting with `/v` & followed by any word characters
  - Taking these 2 together matches, e.g., `helping/vbg`
- `\s+`: 1 or more whitespace characters
- `\w+/v\w*/?`: matches any verb
  - With `\w+`, we match any word, not just `help`
Greediness & Capturing parentheses

- Greediness
  - In Perl, * is *greedy*: it tries to match as much text as possible
    - Consider a text *John goes to the store* and an RE `t.*s`
    - With the normal, greedy *, this matches *to the s*
    - With the non-greedy `*?` (i.e., `t.*?s`), this matches *the s*

- Capturing parentheses: parentheses do more than just distinguish subparts of an RE
  - They “capture” the part(s) of the RE you may want further access to
  - `\b(help\w*/v\w*/?\s+\w+/v\w*/?)\b`
    - We can use $1 to refer to the captured part of the RE (and $2 if there were a second capture, etc.)
    - e.g., `<word>(\w+)</word>` will match the whole string, but only capture the part in-between the XML tags