GRK 2839 Winter School: Corpus & Computational Linguistics

Regular Expressions

Andreas Blombach, Stephanie Evert,
Philipp Heinrich
Lehrstuhl für Korpus- und
Computerlinguistik
https://www.linguistik.phil.fau.de

Friedrich-Alexander-Universität
Philosophische Fakultät und
Fachbereich Theologie

Regular expressions

- Regular expressions (regex/regexp) are a sophisticated wildcard notation from computer science
- Widely used for full-text search (e.g. grep) and by advanced text editors (Emacs, Atom, Notepad++, Kate, …)
  - task: find substring matching pattern in text
- Corpus queries use regular expressions at two levels
  - for matching word forms and annotations (over characters)
    → always matches complete string ≠ full-text search
  - for describing lexico-grammatical patterns (over tokens)
- Different regexp “flavours”: we will use PCRE
  - POSIX, PCRE = Perl-compatible regular expressions, Python, Oniguruma, …

Web interfaces to play around with

- CWB Wordlist Explorer:
  http://corpora.linguistik.uni-erlangen.de/cgi-bin/demos/regex/wordlist_explorer.perl

  match CWB regular expressions against frequency lists of different corpora

- Full-text search & debugging of regular expressions
  - https://regexr.com/
  - https://regex101.com/
  - http://regviz.org/ (for JavaScript, not PCRE)
  - https://www.debuggex.com/

- Play the Regular Expression Crossword
  https://regexcrossword.com/

PCRE: Perl Compatible Regular Expressions

- `(…)?` = optional (0 or 1)
- `(…)*` = any number of repeats (0 or more)
- `(…)+` = at least once (1 or more)
- `(…){3}` = exactly thrice
- `(…){2,4}` = between two and four times (2, 3, 4)
- `(…){4,}` = at least four times
  - Quantifiers refer to the expression that immediately precedes them – if that's supposed to be a pattern of several characters, don't forget to put it in parentheses!
- `(…|…|…)` = alternatives (matches exactly one)
- `.` = any character (matchall)
  - esp.: `.?` (optional character), `.*` (arbitrary string), `.+`
- escapes: `\.` = ., `\*` = *, `\?` = ?, `\+` = +, …

PCRE

- `[aeiou]` = character class (matches exactly one)
  - `[a-z]` = `[abc … z]` and `[A-Z]` = `[ABC … Z]` (NB: no umlauts, <ß> etc.)
  - `[0-9]` = `[0123456789]`
- `[^aeiou]` = everything(!) except `[aeiou]`
- Predefined character classes:
  - `\w` = letters, digits and _ (word character)
  - `\s` = any single whitespace (space, tab, newline, …)
  - `\d` = digit
  - `\pL` = letter, `\p{Ll}` = lower-case letter, `\p{Lu}` = upper-case letter
  - `\pN` = digit, `\p{Cyrillic}` = cyrillic letter, …
    - see https://www.pcre.org/original/doc/html/pcrepattern.html#SEC5

PCRE

Extension for full-text (substring) search:

- not meaningful in corpus queries (which match entire strings)

- **(…)??**, **(…)*?**, **(…)+?**
  = match as few repetitions as possible
  - regular expressions are greedy by default: they try to match as many characters as possible – this behaviour can lead to unexpected and undesirable results
- **^**… = anchor to start of line
- …**$** = anchor to end of line
  - beware: **^(…|…)$** ≠ **^…|…$**
  - in corpus queries, **^** and **$** can usually be used as anchors to the start and end of word forms (reason: one token per line)
- **\b** = anchor matching a “word boundary”
- Cheat sheet for PCRE: https://www.debuggex.com/cheatsheet/regex/pcre

Context-sensitive search: *look-around assertions*

- Find expressions in a specific context without including this context in the result:

…(?=…)	=	*positive look-ahead* (context must follow search expression)
…(?!…)	=	*negative look-ahead* (context must **not** follow search expression)
(?<=…)…	=	*positive look-behind* (context must precede search expression)
(?<!…)…	=	*negative look-behind* (context must **not** precede search expression)

Capturing groups and back-references

- Parentheses create so-called *capturing groups*
  - `(\d{2}):(\d{2})` → groups 1 (hours) and 2 (minutes)
  - can be used for information extraction in e.g. Python or R
  - `(?:…)` = *non-capturing groups* (→ also important to control numbering)
- Back-references to groups: `\1`, `\2`, …
  - `([a-z]+)-\1` → *fifty-fifty*, *wah-wah*, *ack-ack*, …
- Text editors: replacing text with regular expressions (→ *Find and replace*)
  - captured groups can be inserted into replacement text
  - usually with `$1`, `$2`, …
  - “text processing for everybody” (→ *Find in Project*)