Computational Linguistics (Spring 2008, Exercise 2)

Goals

1. Read and tokenize a sample electronic corpus, using regular expressions;
2. practice the mighty `loop()` iteration construct and the PPCRE package;
3. experiment with regular expressions, compile a set of corpus statistics.

1 Bring up the Editor and the LKB Lisp Environment

(a) As always, when logged into one of the IFI Linux machines, launch the LKB grammar development and Lisp environment that we will use throughout the semester, by typing (at the shell prompt):

```
lkb &
```

This will lauch emacs, our editor of choice, with the Lisp and LKB running as sub-processes to the editor. The benefits of this set-up will become apparent sooner or later. In emacs, find the buffer named '`*common-lisp*`'. Here you can interact with Allegro Common Lisp, the Lisp system behind the LKB.

(b) Our course setup should have magically copied a new file '`brown.txt`' into your home directory; open the file in emacs to take a peak at its contents. This is the first 1000 sentences from the historic Brown Corpus, one of the earlier electronic corpora for English. Should you be unable to locate the file, please ask for assistance.

2 Reading the Corpus File; Basic Counts

(a) For reading the contents of the corpus file, remember that it is easy to connect an input *stream* to a file and invoke one of the reader functions, for example:

```
(with-open-file (stream "brown.txt" :direction :input)
  (loop
      for line = (read-line stream nil)
      while line
      append (ppcre:all-matches-as-strings "[^ ]+" line)))
```

Make sure you understand all components of this command; when in doubt, talk to your neighbor or one of the instructors. What is the return value of the above command?

(b) Bind the result of the whole `with-open-file()` expression to a global variable `*corpus*`. What exactly is our current strategy for tokenization, i.e. the breaking up of lines of text into word-like units? How many tokens are there in our corpus?

(c) Write a `loop()` that prints out all tokens in `*corpus*`, one per line. Can you spot examples where our current tokenization rules might have to be refined further, i.e. single tokens that maybe should be further split up, or sequences of tokens that possibly should better be treated as a single word-like unit?

(d) Write an s-expression that iterates through all tokens in `*corpus*` and returns a list of what is called unique word types, i.e. a list in which each distinct word from the corpus occurs exactly once (much like a set). Consider wrapping the `loop()` into a `let()` form, so as to provide a local variable `result` in which the `loop()` can collect unique types, e.g. something abstractly like the following:

```
(let ((result nil))
  (loop
      for token in ...
      when ...
      do ...)
  result)
```

To test whether a token is already contained in `result`, consider writing a recursive function `elementp()`, which takes an *atom* and a *list* as its arguments and returns true if *atom* is an element of *list*. Consider re-using or adapting one of the functions you wrote for the first exercise.

3 Experimenting with Regular Expressions

(a) Grefenstette & Tapanainen (1994), a seminal (albeit somewhat sloppily written) research report on how to tokenize and segment a corpus into sentences, suggest the following rules for the detection of abbreviations:

- a single capital letter, followed by a period, e.g. '`A.`', '`B.`', '`C.`', or '`Å.`';
- a single letter, followed by a period, followed by one or more pairs of a single letter plus period, e.g. '`U.S.`', '`i.e.`', or '`m.p.h.`';
- a capital letter followed by a sequence of consonants and a period, e.g. '`Mr.`' or '`Assn.`'

(b) Translate the above patterns into regular expressions and determine how many of the tokens in `*corpus*` match either of the three descriptions. Note that for this exercise, we are probably only interested in matches for the whole token, i.e. no matches on only a sub-string of a token. Next, combine all three patterns into a single regular expression and determine the total count of candidate abbreviations.

(c) Why is the third condition above restricted to sequences of consonants only? Try a more liberal pattern and determine experimentally what can go wrong with it, i.e. see whether there are tokens that match the new pattern, even though they actually are not abbreviations.

(d) To search for additional abbreviations in `*corpus*`, print out all tokens that end in a period and do *not* match any of the rules above. Can you spot additional examples that should ideally be recognized as abbreviations? See whether you can further refine the above rules, without adding too many false matches (see below).

(e) To improve over the three rules above, Grefenstette & Tapanainen (1994) suggest a search that combines positive and negative evidence from the corpus. Essentially, the idea is to start from a more liberal regular expression, thus creating a set of candidate abbreviations that likely contains what is often called *false positives*: these are matches that are falsely classified as abbreviations, for example words that have a trailing period simply because they occur in sentence-final position. To eliminate false positives, a second round of search through the corpus can look for occurences of each candidate abbreviation without a final period; for a large enough corpus, words that were falsely considered as an abbreviation are likely to also occur in other sentences, typically not followed by a period.

To implement the negative filter, we will need a way of stripping the final period from our candidate abbreviations. One way would be using `ppcre:all-matches-as-strings()` again, but a more straightforward way of extracting a sub-string from a string is the function `subseq()`, for example

```
(subseq "place." 0 5)
→ "place"
```

Note that many Lisp functions can be applied to both lists and strings; among others, this is true for `subseq()` and `length()`.