English hyphenation algorithm in Clojure

Description	Implementing English hyphenation algorithm in Clojure.
Period	Summer 2016
Languages & Libs	Clojure
Tags	Hyphenation, Blog, GitHub, JVM
GitHub	nikonyrh/hyphenator-clj

This is nothing that spectacular (as if anything on my blog is), but I still wanted to describe the outline of the project of porting the *hyphenation* algorithm from PHP to Clojure. The implementation is only about 80 lines of code + comments + 20 lines of unit tests. For comparison the original PHP abomination is about is about 160 LoCs, although it is a bit bloated by implementing the patterns search via a trie data structure instead of using the `strpos` function.

This is the initial step towards re-writing and porting this whole blog platform to Clojure, which was motivated by wanting to learn new language and moving away from PHP projects. I managed to tolerate it for 10 years and got my career started but after working with Python 2 and 3 for three years it seems obvious that PHP will always lack that expressiveness, well-thought design and vision. The only positive side of PHP is the ease of deployment via php-fpm, it took me a while to understand differences between FastGCI and WSGI, and how Nginx + Python API combination had to be set up. Clojure runs on the JVM, which is also widely used in large-scale projects such as Elasticsearch, Neo4j, Apache Hadoop and Spark.

The code starts by reading patterns of english.txt one-by-one, transforming lines like `"_gen3t4"` into `{:str "_gent", :digits {4 3, 5 4}}` where `:digits` is a `hash-map` mapping positions in the string to corresponding integer values. As with the previous implementation, words are pre- and postfixed by underscores and these are used in searched patterns as well.

`match-pattern` function takes a word and a pattern as its arguments and finds all indexes in which the pattern occurs in the word. It then accumulates the maximum observed numerical value for each "slot" (see the article of original implementation for more details). It is implemented by tail-recursing an inner anonymous function until the pattern is not found from the word anymore, at which point it returns the final value. `hyphenate-word` function takes the hyphen and a word, calls `match-pattern` to find all occurrences of patterns, finds indexes of odd values and injects hyphens on positions which don't violate the minimum syllable length of two.

To split sentences into words a `pattern-chars` set is defined, which contains all upper- and lowercase characters which occurred in the patterns. An other utility is the `count-chars` function which takes two strings as its arguments and for each character in the first string it calculates the number of its occurrences in the second string. This is used to count the cumulative number of < and > characters to know if a word is occurring inside `<a html tag>or not</a>`.

The ultimate function which brings all this together is the `hyphenate`. It starts by splitting the sentence into "partitions" (words and word-separators) by using `(partial contains? pattern-chars)`, checks whether odd or even partition indexes are the ones which contain words to-be hyphenated, calculates the cumulative XHTML tag-balance, and merges all these into a `should-hyphenate?` function. Then the partitions are either hyphenated or left as-is and joined back together into a string which is returned.

Writing unit tests was crucial but also extremely interesting discovery process, as `clojure.test` comes with macros which greatly reduce repetition in test case definitions. With the help of self-written `my-are` and `my-deftest` writing tests for functions `is-digit?` and `elem-max` was just two lines of code: `(my-deftest test-is-digit is-digit? \a false \_ false \Z false \0 true \5 true \9 true)` and `(my-deftest test-elem-max elem-max [[-3 1 3] [1 2 -3]] [1 2 3])`.

The convention of this macro is that 1st argument is the test name, 2nd is the tested function, remaining arguments of odd index are input values and at even indexes are expected outcomes. Other important functions are tested in a straight-forward manner as well. I hope this wall of text without any figures was at least a bit interesting documentation, I'm really looking forward to use Clojure in future projects.