Mustache templates in Clojure

Description	Implementing Mustache templates algorithm in Clojure.
Period	Summer 2016
Languages & Libs	Clojure
Tags	Blog, GitHub, JVM
GitHub	nikonyrh/mustache-clj

Mustache is a well-known template system with implementations in most popular languages. At its core it is logicless same templates can be directly used on other projects. For example I am planning to port this blgo engine from PHP to Clojure but I only need to replace LaTeX parsing and HTML generation parts, I should be able to use existing Mustache templates without any modifications. To learn Clojure programming I decided not to use the recommended library but instead implement my own.

I started by looking at existing implementations, most notably Clostache's parser.clj. It is about 380 lines of code, whereas my reference implementation is 115 lines of code. Notably Clostache is heavily type-annotated, uses regular expressions and also implements lambdas. I opted for not having type annotations, not implementing much logic in regexes and not implementing lambdas :)

It was a very exciting moment to finally bring all the pieces together, as the `render` function is implemented as a single pipeline through lexer and parser steps: `(-» template lexer (resolve-partials partials) parser (merge-ast-and-data data) flatten (map escaper) (apply str))`. `lexer` takes an input string as its argument, uses `str-replace` regexes to normalize a few alternative syntaxes, splits it into tokens and adds metadata to them. `resolve-partials` first lexes partials (as they are normal strings) and then iteratively replaces partial references by the referenced values. As partials can refer to other partials this process needs to be repeated until everything has been resolved.

`parser` takes in a sequence of tokens and first adds "path" information to them as Mustache supports nested structures called "Sections". Path is described as a list of names of section-starting nodes between the node and the root. On a second pass tokens are split into partitions based on their path at the current level, essentially building a local view of the abstract syntax tree (AST). Once all that recursion has been done we are left with the complete AST.

The final interesting piece to the puzzle is `merge-ast-and-data` which, as the name suggests, merges AST with the input data. If it encounters a "reference" type token then it is replaced by the corresponding value in input data, on other cases we are dealing with an AST node. If it has a path defined then corresponding data-sequence is loaded from the input and iterated over, recursively calling itself with subsequent AST nodes. If the node doesn't have a path then simply AST nodes are recursively processed. This generates a new AST which is similar to the original but new nodes have been created based on elements of the data.

At this point the tree structure is not needed anymore, thus it is flattened via `flatten`. The only remaining task is to walk over them once more, and checking if they represent a "raw" textual value or if it should be escaped. Escaping is based on walking over string's characters one at a time and checking if their escaped counterpart is found from a hash-map. The final step is to concatenate these strings by `(apply str)`.

This implementation passes all relevat unit tests of Clostache, only lambdas and file-based functions are not implemented at this point. Next steps is to make this "importable" as a library (still thinking whether to put it on Clojars or not). Then I can proceed to implement a LaTeXparser and integrate it with this and my hyphenator-clj projects and I'm one step closer to ditching my PHP-based blog engine. But it is going to be quite a lot of effort as the PHP project has cumulated quite many features and hacks, such as determining file's representative modified date from `git blame`'s output.