Harvey Mudd College
Computer Science 42
Fall 2012

Assignment 1
KWIC Index API using Functional Programming
Due. 11:59 p.m., Wednesday, 19 Sept. 2012

API stands for *Application Programming Interface*, a set of related functions or procedures that perform computational work, exclusive of the user interface. The main function to be implemented is called **`create-kwic-index`**. Its definition is given below. It will be constructed by composing a number of smaller functions, which you will define in the first part of the assignment. I will provide ***unit tests*** (to be described in class on Tuesday) for most of these functions. Submit as file **`a01.rkt`.**

This assignment is about functional programming, so naturally we do not allow non-functional programming. The problems in this assignment are to be done in a ***purely functional style*** using the *Racket* language. Furthermore, because we want to focus on functional composition and higher-order functions, ***use of recursion is also forbidden***, except for the helper functions that I provide. (There will be plenty of recursion next assignment.)

Clearly document your functions. **Be sure to spell each function name in the API exactly as given**, otherwise the grading script will not give you credit for the function.

A "kwic" (**K**ey**W**ord-**I**n-**C**ontext) index is a special index that organizes a list of titles by *keywords* occurring therein. Titles containing a given keyword are grouped together, and the keywords are listed alphabetically. The context consists of the words on either side of the keyword. A given title will appear once in the index for each keyword in it.

Rather than being defined explicitly, *keywords* are defined to be the words occurring in the index that are not *stopwords*, a technical term meaning *noise word*. The user of **`create-kwic-index`** specifies the stop words as a list, as well as a list of punctuation symbols, the use of which will be described subsequently. Accompanying each entry in the index is a *reference number* that enables the user to find more information about the title. In our case, the reference numbers are just the numbers of the articles from the input list, starting with 0. A kwic index for a list of titles, might appear as follows (the italic words are not part of the output). Reading down the the right of the gap shows the keywords alphabetized.

```
                                 gap                          reference number
                                  | |                                        |
overs of this book are too far apart.                                     : 3
           The covers of this book are too far apart.                     : 3
 want less corruption, or more chance to participate in it.               : 5
         My opinions may have changed, but not the fact that I am         : 6
                    ( … abridged for brevity … )
e the ones you don't know very well.                                      : 1
nly normal people are the ones you don't know very well.                  : 1
eighbor as thyself, but choose your neighborhood.                         : 2
```

The index on the previous page was produced from the list of title strings below.

```
(define titles4
 '("It is easier to fight for one's principles than to live up to them."
   "The only normal people are the ones you don't know very well."
   "Love thy neighbor as thyself, but choose your neighborhood."
   "The covers of this book are too far apart."
   "No good deed goes unpunished."
   "I either want less corruption, or more chance to participate in it."
   "My opinions may have changed, but not the fact that I am right."
))
```

Surrounding the word on either side of the gap is the *context*, the words in the title around the keyword, up to the amount of space indicated for each side. On the rightmost end, after the colon is the reference number, which in this case is just the position of the title in the original list, counting from 0. The reference numbers in our case are generated by the program.

A required argument to the index-producing function is a list of ***stopwords***, words on which we do not want to index. Case should be ignored when comparing input words to these. Also, a list of punctuation must be provided, so that a stopword can be checked even if followed by a punctuation mark. These lists are arguments of the **`create-kwic-index`** function. In the current example, the punctuation and list of stop words were specified as follows. These are assumed to be in effect for the test cases below.

```
(define noise4 (map symbol->string '(a am and are as but either for in
                 is it of on or the than that this to too)))

(define punctuation (string->list ".,;:!?"))
```

Requirements: (Do part b. first.)

a. [50 points] Implement function
`(create-kwic-index Stopwords Punctuation Titles)` that returns a list of *triples* (3-element lists), each triple containing:

- the reference number for the title
- a single string of words from the *left* of the keyword onward
- a single string of words to the *right* of the keyword

To simplify this assignment, space in the titles is the only delimiter of words. Any multiple spaces between words are the same as a single space. Punctuation is lumped with the word to which it is connected without intervening spaces.

b. [50 points total] To help you implement function **`create-kwic-index`**, some helper functions can be used, as defined below. These need to be defined to get credit, even if your solution to part a. doesn't use them. However, they are the functions I used in my solution, so I know they work. I provide some tests, but you may want to supply others. **The graders may use tests that are not given here.**

For the first definition, we exploit the function `non-empty-suffixes`, as defined in the freebies file. The definition (which uses recursion) is omitted here, but one unit test is shown.

```
;; Given a list, non-empty-suffixes returns the list of
;; non-empty suffixes of that list, largest to smallest.
;; For example, (non-empty-suffixes '(1 2 3))
;;                          returns '((1 2 3) (2 3) (3)).

 (check-expect (non-empty-suffixes '(1 2 3 4))
                                  '((1 2 3 4) (2 3 4) (3 4) (4)))
```

I give you the solution to the first problem, to illustrate the desired manner of documentation. The meaning of the function is described in comments above. The rationale is in comments below, or on the side.

```
;; Given a list, non-full-prefixes returns the list of the prefixes of
;; that list, other than the list itself, in the order smallest
;; to largest.
;; For example, (non-full-prefixes '(1 2 3 4))
;;           returns '(() (1) (1 2) (1 2 3)).

(define (non-full-prefixes List)
  (cons '()
   (reverse (rest (map reverse (non-empty-suffixes (reverse List)))))))

;; Rationale: Make use of function non-empty-suffixes, which is
;; provided. The non-full prefixes of a list are the reverses of
;; the non-empty suffixes of the original list in reverse,
;; excluding the entire list, but adding the empty list to the front.

(check-expect (non-full-prefixes '(1 2 3 4)) '(() (1) (1 2) (1 2 3)))
(check-expect (non-full-prefixes '(1 2 3)) '(() (1) (1 2)))
(check-expect (non-full-prefixes '(1)) '(()))
```

```
;; is-stopword determines whether Word is in the list StopWords

(define (is-stopword Word StopWords) …)

;; noise4 is defined on page 2
(check-expect (is-true (is-stopword "the"   noise4)) #t)
(check-expect (is-true (is-stopword "The"   noise4)) #t)
(check-expect (is-true (is-stopword "apple" noise4)) #f)
```

```
;; Function mappend expects a function returning a list and a list.
;; It returns a list consisting of the results of the function
;; applied to each element of the list appended together into a single
list.

(define (mappend Function List) …)

;; For testing mappend (could be eliminated with an anonymous function)
(define (zeroTo X) (range 0 (+ 1 X)))

(check-expect (zeroTo 5) '(0 1 2 3 4 5))
(check-expect (mappend zeroTo '(1 2 3)) '(0 1 0 1 2 0 1 2 3))
```

```
;; Given a list, all-cleaves returns the list of all ways of cleaving
;; the list into a prefix and a suffix.
;; For example, (all-cleaves '(1 2 3)) returns
;;              '((() (1 2 3))
;;                ((1)  (2 3))
;;                ((1 2)  (3)))

(define (all-cleaves List) …)

(check-expect (all-cleaves '(1 2 3))
              '((() (1 2 3))
                ((1)  (2 3))
                ((1 2)  (3))))
```

```
;; (ends-with-char String List-of_chars) is #t if String ends with
;; one of the characters in List-of-chars.

(define (ends-with-char String List-of-chars) …)
(check-expect (is-true (ends-with-char "foobar" '(#\r))) #t)
(check-expect (is-true (ends-with-char "foobar" '(#\f))) #f)
```

```
;; Given a non-empty list, all-but-last returns the list of all
;; but the last element. It is an error to call it on an empty list.
;; For example, (all-but-last '(1 2 3 4)) returns '(1 2 3).

(define (all-but-last List) …)

(check-expect (all-but-last '(1 2 3 4)) '(1 2 3))
```

```
;; Given a non-empty string, string-all-but-last returns the string of
;; all but the last character. It is an error to call it on an empty
;; string.

(define (string-all-but-last String) …)

(check-expect (string-all-but-last "foobar") "fooba")
```

```
;; Given a list of strings, split-strings returns a list of lists
;; wherein each list consists of the individual words in an original,
;; string, where words are understood to be separated by one or more
;; spaces. This can rely on the built-in function string-split.

(define (split-strings Titles) …)

(check-expect (split-strings
              '("The rain in Spain"
                "falls mainly in the plain"
                "just plays with my very lazy dog"))

              '(("The" "rain" "in" "Spain")
                ("falls" "mainly" "in" "the" "plain")
                ("just" "plays" "with" "my" "very" "lazy" "dog")))
```

```
;; Given a list of lists, tag-elements returns a list of lists, each
;; consisting of the index of the original inner list (starting from 0)
;; followed by the elements of the list itself.

(define (tag-elements splits) …)

(check-expect (tag-elements '((a) (b c d) (e f)))
                            '((0 a) (1 b c d) (2 e f)))
```

```
;; Given a list consisting of a tag followed by some more lists,
;; distribute-tag returns a list of lists, wherein each of the original
;; lists is tagged with the tag.
;; For example, (distribute-tag '(123 (a b) (c d) (e f)))
;; returns '((123 a b) (123 c d) (123 e f)).

(define (distribute-tag List) …)

(check-expect (distribute-tag '(123 (a b) (c d) (e f)))
                              '((123 a b) (123 c d) (123 e f)))
```

```
;; Given a list of lists, all-tagged-cleaves returns the list of
;; all cleaves of the inner lists, with each cleave tagged with
;; the index of the inner lists position.
;; (proper means that the second list is not empty.)
;; For example (all-tagged-cleaves '((a b) (c d e f) (g h i))) returns
;; '((0 () (a b))
;;  (0 (a) (b))
;;
;;  (1 () (c d e f))
;;  (1 (c) (d e f))
;;  (1 (c d) (e f))
;;  (1 (c d e) (f))
;;
;;  (2 () (g h i))
;;  (2 (g) (h i))
;;  (2 (g h) (i)))

(define (all-tagged-cleaves ListOfLists) …)
```

To use your kwic index in an application, the freebies file supplies a function **`(format Left Right Triples)`** formats the output of function **`create-kwic-index`** into a list of single strings, each of which forms a line of printable index. The variables **`Left`** and **`Right`** indicate the number of characters on the left and right of the gap, respectively. **`Triples`** is a list of triples (lists of three elements), of the kind produced by **`create-kwic-index`**. This was used to create the display on the first page.

Racket built-in functions that could be useful for this assignment

- `map` (including the version that maps 2-argument functions)
- `filter`
- `member` (determines whether first argument is in the second list)
- `sort` (second argument is element-comparison function)
- `reverse`
- `append`

- `string->list` (produces list of characters from a string)
- `string-split` (splits string into list of strings)
- `string-join` (joins separate strings, with interspersed spaces)
- `string-downcase` (produce a lower-case version of the string)
- `lambda` (a special form that creates an anonymous function)

Concepts that might be useful for this assignment

- higher-order functions
- function composition
- functions as arguments
- anonymous functions

References

http://en.wikipedia.org/wiki/Key_Word_in_Context
http://en.wikipedia.org/wiki/Stop_words
http://en.wikipedia.org/wiki/Function_composition_(computer_science)

Design and Development Hints

1. Think about how you are going to decompose the problem before starting to code part a. The functions in part b can serve as hints.

2. Structure your functions as compositions of simpler functions that do specific things.

3. Test functions independently. This makes it much easier to find errors than figuring out what went wrong in the final composition.

4. Test built-in functions when you use them the first time, to make sure that they do what you think they do.

5. Adding extra tests cases never hurt. You may leave them in your submission file.

6. **If anything is unclear, please ask me before investing significant time.**