Harvey Mudd College
Computer Science 60
Fall 2007

Assignment 2
**Applications Using Functional Programming**
Due 11:59 p.m., Monday, Sept 17 2007

All problems in this assignment are to be done in a purely functional style using the Scheme language. Submit two files: one named hw02.scm containing all solutions in scheme, and one named hw02tests.txt containing your tests (described below). **Note that each problem specifies a function to write and what the testing requirements are.** Place both files in a folder called HW2 in your dropbox on Sakai. As always, document your functions clearly. **Be sure to spell each function name exactly as given**, otherwise the automatic grading script might not give you credit for the function.

1. [20 points]
In the context of the scrabble scoring function of Assignment 1, define a function `best-word` that takes a string representing a multi-set (like a set, but repetitions are allowed) of letters and a list of strings representing allowable words, and which returns a word and its score in the list that can be constructed with letters in the multi-set and which has a maximal score among all such words. (If there is a tie, `best-word` returns one of the words arbitrarily.) In grading your problems, we will make sure that the chosen word in each test case is unique.

```
(best-word "academy" (list "ace" "ade" "cad" "cay" "day"))
→ ("cay" 8)

(best-word "apple" (list "peal" "peel" "ape" "pape"))
→ ("pape" 8)

(best-word "pale" (list "peal" "peel" "ape" "pape"))
→ ("peal" 6)
```

Notice that in the second example, two identical letters were used in the multi-set, while in the third example, that word could not be made because of too few p's in the multi-set.

The function `string->list` that takes one argument (a string) and returns a list of characters that make up the string will be useful here (and in the next problem).

**Testing:** For this function you should submit the results of running a few tests besides the ones given above, to prove that your function is, in fact, working. The easiest way to do this is to run several tests in the interpreter, and then save the interpreter window as a text file (Dr. Scheme has an option to do this under "Save Other" in the File menu.)

After you save the tests, you should open the tests file and provide a *brief* annotation about why you chose each test. For example:

```
Test 1: test when the best word is in the middle of the list
```

```
(best-word "academy" (list "ace" "ade" "cad" "cay" "day"))
→ ("cay" 8)
```

You don't need to go overboard here. 3-5 tests will suffice, as long as they test a broad set of conditions.

2. [80 points]
A "kwic" index (**K**ey**W**ord-**I**n-**C**ontext index) is an index that alphabetizes words in a list of titles, so that titles can be looked up based on the occurrence of words in them. A given title may appear multiple times in the index, once for each significant word in the title. Accompanying each entry in the index is a citation number that enables the user to find more information about the title.

Suppose, for example, that our list of titles is the following list of aphorisms (represented as a list of strings):

```
(
"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."
)
```

Then a kwic index for this list of titles, might appear as follows:

gutter reference number

```
                              | |                                |
anged, but not the fact that I am right.                       : 6
overs of this book are too far apart.                          : 3
        The only normal people are the ones you don't know ve  : 1
       The covers of this book are too far apart.              : 3
             Love thy neighbor as thyself, but choose your ne  : 2
            The covers of this book are too far apart.         : 3
 Love thy neighbor as thyself, but choose your neighborhood.   : 2
 My opinions may have changed, but not the fact that I am rig  : 6
 want less corruption, or more chance to participate in it.    : 5
          My opinions may have changed, but not the fact that  : 6
e thy neighbor as thyself, but choose your neighborhood.       : 2
            I either want less corruption, or more chance to   : 5
                          The covers of this book are too fa   : 3
                    No good deed goes unpunished.              : 4
( … abridged for brevity … )
                         Love thy neighbor as thyself, but c   : 2
           Love thy neighbor as thyself, but choose your neigh  : 2
                 It is easier to fight for one's principles    : 0
ight for one's principles than to live up to them.             : 0
ess corruption, or more chance to participate in it.           : 5
e's principles than to live up to them.                        : 0
   The covers of this book are too far apart.                  : 3
             No good deed goes unpunished.                     : 4
 one's principles than to live up to them.                     : 0
le are the ones you don't know very well.                      : 1
                      I either want less corruption, or more   : 5
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
                              | |                                |
                              gutter reference number
```

Notice the whitespace *gutter* down the middle of the index. The words to the right of the gutter are those on which alphabetization has occurred. Surrounding the word on either side is the *context*, the rest of the title, up to the amount of space provided. 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.

A required argument to the index-producing function is a list of ***noise words***, words on which we do not want to index. In the current example, the list of noise words was specified as:

```
("a" "A" "from" "From" "in" "In" "of" "Of" "on" "On" "the" "The")
```

**Code Requirements:**

At a high level, we will break the problem into two parts: the problem of finding the strings in the kwic, and the problem of formatting the kwic for readability. Thus, *two* functions are required to be submitted:

Function **`(kwic Noise Titles)`** returns a raw list of triples containing:

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

(Note that we swap the position of left and right to make visual debugging simpler. The proper order is restored by the **`format`** function below.) For purposes of this assignment, any multiple spaces between words are replaced by a single space. Also, we take the liberty of including punctuation marks with the word before it. In other words, space in the titles is the only delimiter of words, not punctuation marks.

Function **`(format Left Right Triples)`** formats the output of **`kwic`** into a list of single strings, each of which forms a line of printable index. The variables **`Left`** and **`Right`** indicates the number of spaces on the left and right of the gutter, respectively. Triples is a list of triples, such as produced by **`kwic`**. **`format`** does not put in ends-of-line characters explicitly. That is left to the ultimate print procedure, about which we do not worry in this assignment.

One reason for separating out **`kwic`** as a function is that we might make further use of the index contents in list form rather than string form, for example in developing a larger implementation. This gives us an "API" (Application Programming Interface) for the problem. The **`format`** function just gives one possible output format of the information.

In the following test examples, I am working with a list of abbreviated titles for brevity. The test call of kwic is:

```
(kwic
  (list "The" "to" "is" "It" "thy")  ; noise
  (list                              ; titles
   "It is easier to fight."
   "The only normal people."
   "Love thy neighbor."
   )
 )
```

The result of that call, *where spaces have been inserted manually only for readability*, is:

```
'(("easier to fight."    "It is"            0)
  ("fight."              "It is easier to"  0)
  ("Love thy neighbor."  ""                 2)
  ("neighbor."           "Love thy"         2)
  ("normal people."      "The only"         1)
  ("only normal people." "The"              1)
  ("people."             "The only normal"  1))
```

The result of a call to **`format`** on the above list of triples with **`Left`** = 10 and **`Right`** = 20 is shown below.

```
("     It is easier to fight.     : 0"
 " easier to fight.               : 0"
 "           Love thy neighbor.   : 2"
 "  Love thy neighbor.            : 2"
 "  The only normal people.       : 1"
 "       The only normal people.  : 1"
 "nly normal people.              : 1")
```

The above result was technically produced by the following call, which uses `kwic` to generate the list of triples:

```
(format 10 20
 (kwic
  (list "The" "to" "is" "It" "thy")
  (list
   "It is easier to fight."
   "The only normal people."
   "Love thy neighbor."
   )
  )
)
```

**Testing Requirements:**

For part 2 in your testing file, you only need to include a short paragraph (a few sentences) about why the above example is or is not a good test case. Are there any conditions missing from the above example that you would want to test?

**Scheme built-in functions that could be useful for this assignment**

- `lambda` (a form that creates a function)
- `map`
- `foldr`
- `sort` (second argument is element-comparison function)
- `string->list`
- `list->string`
- `number->string` (to enable numbers to be concatenated with strings)
- `append`
- `reverse`
- `string-append`
- `string-length`
- `substring`
- `string-ci<?` (compares strings case-insensitively)
- `string-ci=?` (compares strings case-insensitively)

**User-defined functions that could be useful for this assignment**

- `keep`
- `drop`

**Concepts that could be useful for this assignment**

- functions as arguments
- anonymous functions
- accumulator arguments

**Design and Development Hints**

1. Structure your functions as layered applications of simpler functions that do specific things.

2. Test the simpler functions independently. This is much easier than figuring out what went wrong in the final composition.

3. Test built-in functions before you use them, to make sure that you know what they do.

4. Think about how you are going to decompose the problem before coding—down the level of individual Scheme function wherever possible.