Obligatory exercise 2a (INF4820, Fall 2013)

This is part 1:2 of the second obligatory exercise in INF4820 2013. You can obtain up to 10 points for 2a, and you need a minimum of 12 points for 2a + 2b in total. Please make sure you read through the entire problem set before you start coding (4 pages). If you have any questions, send an email to `inf4820-help@ifi.uio.no`.

Answers must be submitted via Devilry by the end of the day (23:59) on Sunday, September 29th. Please provide your code and answers (in the form of Lisp comments) in a single '.lisp' file.

Summary of goals for this exercise

- Implement a vector space model for representing semantic / distributional word similarity.
- Design data structures for representing high-dimensional sparse feature vectors.
- Implement functionality for computing the similarity of words based on the distance of their (length-normalized) feature vectors in the space.

Files you'll need

We will again be using a subset of the Brown corpus as our data set (but a larger chunk than in the previous assignment). You can copy the file '`brown20000.txt`' to your IFI home directory with the following shell command:

```
cp ~erikve/inf4820/brown20000.txt ~/
```

The data consists of 20.000 sentences as plain text, one sentence per line. (Please do not re-distribute, as the data is licensed.) In addition, you will need the list of words given in the file '`words.txt`', with one word per line (122 all together). These are the words we are going to model. Grab the file here:

```
cp ~erikve/inf4820/words.txt ~/
```

1 Theory: Word–context vector space models

The theoretical background for this problem set is what we covered in the previous lecture (11/9). Our goal is to construct a vector space model where words are represented as *feature vectors* in a high-dimensional *feature space*.

For a given word, we will define the features to be the other words co-occurring with it in the contexts found in our corpus ('`brown20000.txt`'). (Recall that, each feature will be assigned to a distinct dimension in the space.) The idea is to model semantic similarity in terms of distributional (or contextual) similarity, and in turn to model this as geometrical distance in the vector space. We will here take a crude *bag-of-words* (BoW) approach to how we define context: The features we extract for a given occurrence of a word will simply consist of all other words co-occurring within the same sentence. Note that, in this exercise we will only construct feature vectors for the m words specified in the file 'words.txt' (described above). (In practice this also means that we will basically be ignoring sentences that doesn't contain any of the words in the list.)

(a) In a few sentences, discuss other ways of how we could have defined the notion of context here.

2 Creating the vector space

(a) To represent the vector space in our Lisp code, we will implement an abstract data type that we call `vs`. Using `defstruct`, define a structure[1] `vs` that has at least the following slots;

- `string-map`; For mapping strings to numeric identifiers.
- `id-map`; For mapping numeric identifiers to strings.
- `matrix`; The collection of feature vectors can, abstractly, be thought of as an $m \times n$ matrix, where each of our m feature vectors correspond to a row in a matrix. The columns then correspond to the n dimensions in the feature space. We will sometimes refer to this matrix as the *co-occurrence matrix*, and this will store the frequency counts for our model.
- `similarity-fn`; A function for computing word similarity (in terms of the proximity of their feature vectors in the space).

(b) In this exercise you will define a function `read-corpus-to-vs` that reads the content of the corpus (one sentence per line), and populates the matrix of a `vs` structure with co-occurrence counts. (As described below, however, our reader function will require several other helper functions as well.) The matrix should contain one row per word (corresponding to the words in 'words.txt'), and one column per feature. Each element e_{ij} represents the number of times that word i and feature j have co-occurred in the same sentence. (Note, however, that with our particular BoW-definition of context, the features are themselves also just words.) Example function call:

```
CL-USER(7): (setf space (read-corpus-to-vs "brown20000.txt" "words.txt"))
#S(VS :STRING-MAP ...
      :ID-MAP ...
      :MATRIX ...
      :SIMILARITY-FN ...
      ... )
```

When working with large models we typically want to work with *numerical identifiers* rather than directly with strings. This often means we can store and index the data more compactly and efficiently. When populating the co-occurrence matrix you should therefore assign numerical identifiers to strings, taking care to also keep track of the reverse mapping so we can get from ids and back to strings again. The slots `string-map` and `id-map` in our `vs` structure are intended to store this information. Write functions `string2id` and `id2string` for mapping strings to ids and *vice versa*. For generality, you should maintain separate such mappings for words and features (you might consider storing multiple mapping "tables" in the slots `string-map` and `id-map`).

You should think carefully about the choice of data-structure for representing the co-occurrence matrix (i.e. the collection of feature vectors). Keep in mind that the feature vectors will typically be very sparse. Each word will give rise to a separate feature, and each feature represents a dimension in the vector space. However, only a few of the features will typically be 'active' (i.e. non-zero) for each word. Do Lisp vectors provide the best way to represent our sparse feature vectors? As part of you answer to problem 2b, please include a few sentences motivating your choice of data structure for the matrix / feature vectors.

When reading in the words from the Brown corpus, we will also want to do some simple *text normalization* to reduce the 'noise' and give us fewer unique word types. You should at least make sure that the words are all lower-case and without any attached punctuation (in prefix/suffix position). Built-in functions like `string-downcase` and `string-trim` come in very handy for this. Write a function `normalize-token` to take care of this (taking an individual word string as input argument).

As discussed in the lecture slides and in J&M, we typically also want to filter out very high frequency words from our features, such as closed-class function words. You can use the following (quite arbitrary and unscientific) list for filtering out such 'stop-words'. Of course, feel free to compile you own stop-list instead. Tip: For more efficient look-up of words in the stop-list, consider using a hash-table instead of a list.

[1] Note that Seibel 2005 doesn't cover "structs" — to see examples of how to use `defstruct` see e.g., the lecture notes from 4/9 or the hyperspec: `http://www.lispworks.com/documentation/HyperSpec/Body/m_defstr.htm`

```
(defparameter *stop-list*
    '("a" "about" "also" "an" "and" "any" "are" "as" "at" "be" "been"
      "but" "by" "can" "could" "do" "for" "from" "had" "has" "have"
      "he" "her" "him" "his" "how" "i" "if" "in" "is" "it" "its" "la"
      "may" "most" "new" "no" "not" "of" "on" "or" "she" "some" "such"
      "than" "that" "the" "their" "them" "there" "these" "they" "this"
      "those" "to" "was" "we" "were" "what" "when" "where" "which"
      "who" "will" "with" "would" "you"))
```

(c) Write a function that retrieves the feature vector for any given word in our list, e.g.;

```
(get-feature-vector space "food")
```

(d) Write a function `print-features`, taking three arguments; a `vs` structure, a word (as a string) and a number k. The function should then print a sorted list of the k features with the highest count/value for the given word. Example function call (your results may differ):

```
CL-USER(10): (print-features space "university" 10)
university 26
college 15
dr 15
state 14
work 12
emory 12
applications 12
professor 11
students 11
study 10
NIL
```

3 Vector operations

(a) The Euclidean norm or length of a vector $\vec{x}$ is defined as follows:

$$\|\vec{x}\| = \sqrt{\sum_{i=1}^{n} \vec{x}_i^2}$$

Write a function `euclidean-length` that computes the norm of a given feature vector.

(b) It is often desirable to work with so-called *unit vectors* or *length normalized* vectors. (One important reason for this is that we want to reduce bias effects caused by e.g. skewed frequency distributions. It also makes it possible to compute similarity functions such as the cosine much more efficiently). A vector has unit length if its Euclidean norm is 1:

$$\|\vec{x}\| = \sqrt{\sum_{i=1}^{n} \vec{x}_i^2} = \sum_{i=1}^{n} \vec{x}_i^2 = 1$$

Write a function `length-normalize-vs` taking a `vs` structure as input and then destructively modifying its co-occurrence matrix so that all the feature vectors have unit length. This should have an effect similar to the following:

```
CL-USER(13): (euclidean-length (get-feature-vector space "boston"))
27.820856
CL-USER(14): (length-normalize-vs space)
#S(VS ... )
CL-USER(15): (euclidean-length (get-feature-vector space "boston"))
1.0000001
```

(c) The cosine measure, defined as

$$\cos(\vec{x}, \vec{y}) = \frac{\sum_i \vec{x}_i \vec{y}_i}{\sqrt{\sum_i \vec{x}_i^2} \sqrt{\sum_i \vec{y}_i^2}} = \frac{\vec{x} \cdot \vec{y}}{\|\vec{x}\| \|\vec{y}\|}$$

is perhaps the most commonly used similarity measure in vector space models. When working with *length normalized* vectors, the cosine can be computed simply as the *dot-product* (aka the inner product). The dot-product of two vectors $\vec{x}$ and $\vec{y}$ is defined as:

$$\vec{x} \cdot \vec{y} = \sum_{i=1}^{n} \vec{x}_i \vec{y}_i$$

Write a function `dot-product` that computes this similarity score for two given feature vectors. Store the function in the slot `similarity-fn` of our vector space structure.

(d) Finally, write a function `word-similarity` that computes the similarity of two given words in our model. The function should take three input parameters; a `vs` structure, and two words (as strings). The function should then look up the corresponding feature vectors and compute their similarity according to the similarity function stored in the `vs` structure. Example of a function call (again, your exact return values need not be identical):

```
CL-USER(24): (word-similarity space "university" "college")
0.55649346
CL-USER(26): (word-similarity space "university" "bread")
0.12200105
```

Happy coding!