Obligatory assignment 1 (INF4820, Fall 2012)

This is the first out of five obligatory assignments in INF4820 for the fall 2012. The assignments won't count in the final grade, but all have to be passed in order to qualify for the final exam. More important than getting everything right is to make sure you show that you've tried: If there is a problem you find you can't solve, write down your thoughts on where/why you got stuck. If you find you cannot provide code, try to explain in words how you believe the problem could be solved.

To get started with using Common Lisp and Emacs, and connecting to an IFI Linux server, see the guide posted on the course web page:
`http://www.uio.no/studier/emner/matnat/ifi/INF4820/h12/undervisningsmateriale/get-started.pdf`

Submitting: Please submit your answers through Devilry by the end of the day (23:59) on Thursday, September 6th: `https://devilry.ifi.uio.no/`. At the group session the following Friday (Sept. 7th) we will walk you through a suggested solution to the problem set and also release the 2nd assignment. Please provide all the answers in a single '.lisp' file, including your code and answers (in the form of Lisp comments). Note that it is also good practice to generously document your code with comments (the Lisp reader will ignore everything following a semicolon / ';').

1 List processing

From each of the following lists, select the element `pear`:

(a) `(apple orange pear lemon)`

(b) `((apple orange) (pear lemon))`

(c) `((apple) (orange) (pear) (lemon))`

(d) Show how the lists (b) and (c) above can be created through nested applications of the `cons` function. To illustrate, for the list in example (a):

```
(cons 'apple (cons 'orange (cons 'pear (cons 'lemon nil))))
```

(e) Assume that the symbol `*foo*` is bound to a long list of unknown length, e.g. `(a b c ... x y z)`. Try to show a few different approaches for selecting the next-to-last element of `*foo*`.

2 Interpreting Common Lisp

What is the purpose of the following function? How does it achieve that goal?

```
(defun ? (?)
  (if (consp ?)
      (cons (? (first ?))
            (? (rest ?)))
     ?))
```

Note: Please comment specifically on the various usages of the symbol '`?`' in the function definition.

3 Variable assignment

Fill in the missing s-expressions below (i.e. replace '`????`') so that all expressions will evaluate to `42`.

(a)
```
(let ((foo (list 0 42 2 3)))
  ????
  (first foo))
```

(b)
```
(let* ((keys '(:a :b :c))
       (values '(0 1 2))
       (pairs ????))
  ????
  (rest (assoc :b pairs)))
```

(c)
```
(let ((foo (make-hash-table)))
  (setf (gethash 'meaning foo) 41)
  ????
  (gethash 'meaning foo))
```

4 Recursion and iteration

(a) Write a recursive function that counts the number of times that a given symbol appears as an element in a list.

```
(defun count-member (symbol list)
  ...)
```

Example of expected behaviour:
`? (count-member 'c '(c a a c a c))` → 3.

(b) Now write a non-recursive function that does the same thing (for example by iterating over the list using `dolist` or `loop`).

5 Reading a corpus file; basic counts

For this exercise you need the file '`brown1000.txt`' which you can copy to your own home directory from '`~erikve/inf4820/brown1000.txt`'. To grab a copy you can use the terminal command
'`cp ~erikve/inf4820/brown1000.txt ~/`'
The file contains the first 1000 sentences from the historic Brown Corpus, one of the first electronic corpora for English.

To break up each line of text from the corpus file into a list of tokens (word-like units), we suggest the following function. Make sure to understand the various '`loop`' constructs used here, and also look up the descriptions of '`position`' and '`subseq`', to work out how this function works.

```
(defun tokenize (string)
  (loop
   for start = 0 then (+ space 1)
   for space = (position #\space string :start start)
   for token = (subseq string start space)
   unless (string= token "") collect (subseq string start space)
   until (not space)))
```

For reading the contents of the corpus file, remember that it is easy to connect an input *stream* to a file and invoke one of the reader functions, for example:

```
(with-open-file (stream "brown1000.txt" :direction :input)
  (loop
      for line = (read-line stream nil)
      while line
      append (tokenize line)))
```

(a) Make sure you understand all components of this expression; when in doubt, talk to your neighbor or one of the instructors. Can you describe the return value of the above expression?

(b) Bind the result of the whole `(with-open-file ...)` expression to a global variable `*corpus*`. How many tokens are there in our corpus?

(c) What exactly is our current strategy for tokenization, i.e. the breaking up of lines of text into word-like units? Inspecting the contents of `*corpus*`, can you spot examples where our current tokenization strategy might be further refined, e.g. single tokens that maybe should be further split up, or sequences of tokens that possibly would better be treated as a single word-like unit?

(d) Write an s-expression that iterates through all tokens in `*corpus*` and returns a hash-table where the *keys* are the unique word types (i.e., each distinct word from the corpus corresponds to a hash key) and where the *values* count the corresponding occurrences (i.e., the number of times the word is found in `*corpus*`).

(e) How many unique word types are there in `*corpus*`?