INF4820; Fall 2016: Obligatory Exercise (1)

Goals

1. Become familiar with emacs and the Common Lisp interpreter;
2. practice basic list manipulation: selection, construction, predicates;
3. write a series of simple (recursive) functions; compose multiple functions;
4. read and tokenize a sample corpus file, practice the mighty `loop`.

Background

This is the first out of three obligatory exercises (and the first out of five problem sets) in INF4820. All three exercises must be passed to qualify for the final exam. In order to pass this first exercise, you need to obtain minimally six points out of a maximum of ten. Please see the course web page for more details on how the exercises are organized and for the strict nature of our deadlines:
`http://www.uio.no/studier/emner/matnat/ifi/INF4820/h16/exercises.html`

If you have not already done so, have a look at the *Common Lisp Set-Up* guide on how to get up and running with Common Lisp, which is linked from the course page.

Submitting Solutions must be submitted through Devilry by noon (12:00) on Wednesday, September 14: `https://devilry.ifi.uio.no/`. Please provide your solution as a single '`.lisp`' file, including your code and answers (in the form of Lisp comments). Please also generously document your code with comments (the Lisp reader will ignore everything following a semicolon '`;`').

1 List Processing

From each of the following lists, show code examples for selecting the element `pear`:

(a) `'(apple orange pear lemon)`

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

(c) `'((apple) (orange) (pear))`

(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 (but with at least two or more elements), e.g. `'(a b c ... x y z)`. Show a few different approaches for selecting the next-to-last element of `*foo*`. As always with programming, there are many ways to do it – try using both built-in functions and writing your own.

2 Interpreting Common Lisp

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

```
(defun foo (foo)
  (if foo
      (+ 1 (foo (rest foo)))
      0))
```

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

3 Variable Assignment

In this exercise we will try to get familiar with how to modify values in various data structures (more specifically lists, association lists, hash tables and arrays). Fill in the missing s-expressions below (i.e. replace '`????`') so that all the `let`-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))
```

(d)
```
(let ((foo (make-array 5)))
     ????
     (aref foo 2))
```

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. Actually, there already is a built-in sequence function called `count` that does the same thing, but we assume, of course, that you not use it.

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

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

As a bonus (optional), see if you can write the function using tail-recursion in addition to 'plain' recursion.

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

5 Reading a Corpus File: Basic Counts

For this exercise you need the file '`brown1.txt`' which is part of the SVN-controlled code and data repository for the class. Assuming you have successfully completed Step (1) from the setup instructions for the course (see above), the file should be located at the path '`~/inf4820/1/brown1.txt`' (if not, please synchronize your local copy with the repository, using a command like '`svn update ~/inf4820`') . The file contains (a small part of) the historic Brown Corpus, one of the first electronic corpora for English. As a note on terminology, by *corpus* (plural *corpora*) we simply mean a large collection of digital texts.

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 token
      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) The term *tokenization* refers to the task of breaking up of lines of text into individual words. What exactly is our current strategy for tokenization? 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 a given word is found in `*corpus*`).

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

(f) Refine our tokenization function above to downcase all string and strip initial or trailing punctuation marks from each token; to determine characters that should be stripped off token strings, consider the pre-defined predicate `alphanumericp`. How does the count of unique word types change, and what is the most frequent token when using the original (naïve) vs. the refined approach to tokenization?