Algorithms for AI and NLP (Fall 2014), Assignment (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 of three obligatory exercises (and the first of five problem sets that will need to be submitted) in INF4820. In order to pass this exercise, you need to obtain at least six points out of a maximum of ten available points. Please see the course web page for additional information on how the obligatory exercises are organized:

`http://www.uio.no/studier/emner/matnat/ifi/INF4820/h14/exercises.html`

If you have not done so already, please first work through our 'getting started' instructions on the course page, to obtain the basic setup for interactive Common Lisp development in this course:

`http://www.uio.no/studier/emner/matnat/ifi/INF4820/h14/setup.html`

All solutions must be submitted through *Devilry* by the end of the day (23:59) on Wednesday, September 10; anytime between now before the submission deadline, please upload your results to:

`https://devilry.ifi.uio.no/`

For this exercise, please provide your solution as a single '`.lisp`' file, including your code, answers, and explanations. To make the file a valid Lisp program, please format non-code answers and explanations as Lisp comments (lines prefixed by one or more semicolons). For full credit, please make it a habit to generously document your code with comments.

1 Emacs Basics

If you have not used emacs before, please use a few minutes (or more) to work through its tutorial ('`C-h t`'), print the *reference card* linked from the course page, and keep it under your pillow.

(a) In part because it was conceived before graphical user interfaces gained popularity, emacs at times defines its own terminology. What are the emacs terms for what is commonly called *cut* and *paste* (removing a block of text, storing it in the clip board, and inserting a block of text from the clip board, respectively)?

(b) What is the key binding to start what emacs calls an *incremental search*, i.e. searching forward in the current buffer for a specific pattern? How does one advance the search to subsequent or previous matches of the pattern?

(c) Name at least five different ways in SLIME for sending Common Lisp code from an emacs buffer (that shows the contents of a file) to the Common Lisp environment for execution; explain briefly what each one does.

2 List Manipulation

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 is so often the case in programming, there are many ways to do it—try using both built-in functions and writing your own.

3 Interpreting Common Lisp

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

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

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

4 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))
```

5 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. There actually is a built-in sequence function called `count`, that does the same thing, but for this exercise you must not use it.

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

Following is an example call and expected result:

`? (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 `dolist` or `loop`).

6 Reading a Corpus File: Basic Counts

For this exercise you need the file '`brown.txt`', which is part of the SVN-controlled code and data repository for the course. 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/etc/brown.txt`'. 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 "~/inf4820/etc/brown.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*`? What are the five most frequently used tokens?