Algorithms for AI and NLP (Fall 2009, Exercise 1)
— Part A —

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()`.

1 Bring up the Editor and the Allegro Common Lisp Environment

(a) For our practical exercises, we will be using Allegro Common Lisp (ACL), which is installed on all Linux machines at IFI. The machines in our laboratory room and many of the IFI students laboratories are installed with Linux, and it is sufficient to just log in and start with step (b) below.

When working from home or one of the IFI student laboratories with Windows installations, one needs to connect to a Linux machine first—using the X Window System to allow remote applications to display on the local screen. The standard IFI Windows image includes a link on the desktop (labeled '*xterm*') to launch an X server and connect to a Linux machine. When working from home, one can either use Windows remote desktop to first connect to the IFI Windows universe or ssh(1) (the secure remote shell protocol) with X forwarding.

For a remote desktop session, connect to '`windows.ifi.uio.no`', log in, and follow the instructions for Windows above; note that Windows remote desktop clients are available for Linux and MacOS too, and the remote desktop protocol appears to make quite effective use of lower-bandwidth network connections.

To use ssh(1), connect to the Linux server '`login.ifi.uio.no`' and make sure to request X forwarding; this is achieved by virtue of the '`-Y`' command line option to the ssh(1) client.

(b) To prepare your account for use in this class, you need to perform a one-time configuration step. At the shell command prompt (on one of the IFI Linux machines), execute the following command:

```
~oe/bin/inf4820
```

This command will add a few lines to your personal start-up file '`.bashrc`'. For these settings to take effect, you need to log out one more time (from the Linux session at IFI only, not necessarily your Windows or other session at home, if working remotely) and then back in. Once complete, you need not worry about this step for future sessions; the additions to your configuration files are permanent (until you revert them one day, maybe towards the end of the semester).

(c) Start the integrated Lisp development environment that we will use throughout the semester, by typing (at the shell prompt):

```
acl &
```

This will lauch emacs, our editor of choice, with Allegro Common Lisp running as sub-processes to the editor. The benefits of this set-up will become apparent sooner or later. In emacs, find the buffer named '`*common-lisp*`'. Here you can interact with the Lisp interpreter, i.e. type in s-expressions at the prompt, have them evaluated, and the result returned to you. Our default setup makes use of the Allegro CL emacs – Lisp interface (ELI), which is similar in spirit to the open-source SLIME (as assumed by Seibel, 2005). Note that, unfortunately, ELI and SLIME differ in how they bind commands to keys in emacs; to get started, we will assume ELI as our primary development environment. Once we look closer on the functionality of our integrated Lisp development environment, we will have more to say on both ELI and SLIME.

For this session, we do not supply a starting package or skeleton software, but will mostly interact with the Lisp interpreter directly, i.e. evaluate Lisp expressions through the '`*common-lisp*`' buffer. To record your results for later inspection, in emacs, consider constructing a file in which you save your results and comments. For the final part of this assignment, however, it is advisable to compile your Lisp code before running it, as some (recursive) functions will run many orders of magnitude faster when compiled Please see the *Emacs and Lisp Cheat Sheet* available from the course page.

2 List Selection

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

(a) `(apple orange pear lemon)`

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

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

3 List Construction

Show how the second and third lists from exercise 2 (i.e. (b) and (c)) can be created through nested applications of `cons()`, e.g.

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

for the first example.

Note: In the notation '`cons()`' the pair of parentheses is not part of the operator name but merely indicates that the symbol is used as a function name.

4 Variable Binding and Evaluation

What needs to be done so that each of the following expressions evaluate to 42?

(a) `foo`

(b) `(* foo bar)`

(c) `(length baz)`

(d) `(length (rest (first (first (reverse baz)))))`

5 Quoting

Determine the results of evaluating the following expressions and explain what you observe.

(a) `(length "foo bar baz")`

(b) `(length (quote "foo bar baz"))`

(c) `(length (quote (quote "foo bar baz")))`

(d) `(length '(quote "foo bar baz"))`

6 List Selection

Assume that the symbol `*foo*` is bound to a looong list of unknown length, e.g. `(a b c ... x y z)`.

(a) Find a way of selecting the next-to-last element of `*foo*`.

(b) Are there other expressions that achieve the same effect and use a method of selection that is different from your solution to exercise (a) in an interesting way?

7 Interpreting Common Lisp

What is the purpose of the following function; how does it achieve that goal? Explain the effect of the function using (at least) one example.

```
(a) (defun ? (?)
      (append ? (reverse ?)))
```

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

8 A Predicate

Write a unary predicate `palindromep()` that tests its argument as to whether it is a list that reads the same both forwards and backwards, e.g.

```
? (palindromep '(A b l e w a s I e r e I s a w E l b a))
→ t
```

9 Recursive List Manipulation

(a) Write a two-place function `where()` that takes an atom as its first and a list as its second argument; `where()` determines the position (from the left) of the first occurrence of the atom in the list, e.g.

```
? (where 'c '(a b c d e c))
→ 2
```

Note: Like all Common Lisp functions using numerical indices into a sequence, `where()` counts positions starting from `0`, such that the third element is at position `2`.

(b) Write a two-place function `ditch()` that takes an atom as its first and a list as its second argument; `ditch()` removes *all* occurrences of the atom in the list, e.g.

```
? (ditch 'c '(a b c d e c))
→ (A B D E)
? (ditch 'f '(a b c d e c))
→ (A B C D E C)
```

10 More Recursion

(a) Write a two-place function `set-union()`, that takes as its arguments two sets (represented as lists in which no element occurs more than once and the order of elements is irrelevant); not so surprisingly, `set-union()` returns the union of the two input sets. Analogously, write two-place functions `set-intersection()` und `set-subtraction()`, which compute the intersection and set difference, respectively; e.g.

```
? (set-union '(a b c) '(d e a))
→ (C B D E A)
? (set-intersection '(a b c) '(d e a))
→ (A)
? (set-subtraction '(a b c) '(d e a))
→ (B C)
```

Note: All three functions can assume that their input arguments are proper sets. Consider using functionality implemented earlier during this exercise for re-use in (at least) the definition of `set-subtraction()`.

11 Reading a Corpus File; Basic Counts

Our course setup should have magically copied a new file ‘`brown.txt`’ into your home directory; open the file in emacs to take a peak at its contents. This is the first 1000 sentences from the historic Brown Corpus, one of the earlier electronic corpora for English. Should you be unable to locate the file, please ask for assistance.

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 (unless (null end) (+ end 1))
      for end = (unless (null start) (position #\space string :start start))
      while start collect (subseq string start end)))
```

(a) 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 "brown.txt" :direction :input)
  (loop
      for line = (read-line stream nil)
      while line
      append (tokenize line)))
```

Make sure you understand all components of this command; when in doubt, talk to your neighbor or one of the instructors. What is the return value of the above command?

(b) Bind the result of the whole `with-open-file()` expression to a global variable `*corpus*`. What exactly is our current strategy for tokenization, i.e. the breaking up of lines of text into word-like units? How many tokens are there in our corpus?

(c) Write a `loop()` that prints out all tokens in `*corpus*`, one per line. Can you spot examples where our current tokenization rules might have to be refined further, i.e. single tokens that maybe should be further split up, or sequences of tokens that possibly should better be treated as a single word-like unit?

(d) Write an s-expression that iterates through all tokens in `*corpus*` and returns a list of what is called unique word types, i.e. a list in which each distinct word from the corpus occurs exactly once (much like a set). Consider wrapping the `loop()` into a `let()` form, so as to provide a local variable `result` in which the `loop()` can collect unique types, e.g. something abstractly like the following:

```
(let ((result nil))
  (loop
      for token in ...
      when ...
      do ...)
  result)
```

To test whether a token is already contained in `result`, consider writing a recursive function `elementp()`, which takes an *atom* and a *list* as its arguments and returns true if *atom* is an element of *list.* Consider re-using or adapting one of the functions you wrote earlier.

(d) Use a hash table to obtain word counts, i.e. the number of occurences for each unique word type.

12 Submitting Your Results

This first obligatory assignment has two parts, and the second part will be provided through the course web site in early September. However, we require that you submit your results individually for each of the two parts, where each part accounts for a maximum of fifty points. Please submit your results for Part A in email to Erik (‘`erikve@ifi.uio.no`’) and Gordana (‘`gordanil@ifi.uio.no`’) before the end of the day on Thursday, September 10. Ideally, please provide a text file, including your code and answers (in the form of Lisp comments) to the questions above. Note that it is good practice to generously document your code with comments (using the '`;`' character, making the Lisp system ignore everything that follows the semicolon).