Computational Linguistics (Spring 2009, 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()`.

1 Bring up the Editor and the Allegro Common Lisp Environment

(a) For our practical exercises, we will be using Allegro Common Lisp (ACL) and a specialized software tool called the Linguistic Knowledge Builder (LKB). Both are installed on all Linux machines at IFI, and because the LKB is implemented in Lisp, we will always be working through the LKB environment, even when just programming. Many of the IFI students laboratories are installed with Linux, where it would be sufficient to just log in and start with step (b) below.

However, when working from our current laboratory or from home, 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 '*Linux*') to launch an X server and connect to another machine, running Linux. Double-click the *Linux* icon and login to the remote 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), one can connect to the Linux server '`login.ifi.uio.no`' and request so-called X forwarding, i.e. allow programs running on the remote Linux server to display locally. This is achieved by virtue of the '`-Y`' command line option to the ssh(1) client in current versions, or by virtue of '`-X`' in older versions.

(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/inf2820
```

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 your Windows session or login 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) Launch the LKB grammar development and Lisp environment that we will use throughout the semester, by typing (at the shell prompt):

```
lkb &
```

This will start emacs, our editor of choice, with the Lisp and LKB 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 Allegro Common Lisp, the Lisp system behind the LKB.

For this session, we do not supply a starting grammar or skeleton software, but will merely interact with the Lisp interpreter directly, i.e. evaluate Lisp s-expressions through the '`*common-lisp*`' buffer. To record your results for later inspection, in emacs, you can construct a file in which you save your results and comments. If you are comfortable with emacs already (or are feeling courageous), feel free to start using the more advance interface between emacs and the LKB and Lisp system. For example, rather than typing directly into the Lisp prompt (through the emacs '`*common-lisp*`' buffer), you might put all your

code into a file '`exercise1.lsp`' from the beginning. You can then use emacs commands like '`M-C-x`' (evaluates the top-level s-expression currently containing the cursor) or '`C-c C-b`' (evaluates the entire buffer) to interactively load your code into the Lisp system; remember that, after each change you make in the file '`exercise1.lsp`', you need to re-evaluate the code before your changes will take effect. For testing purposes, we still recommend you use the '`*common-lisp*`' Lisp prompt.

(d) If, while reading the instructions above, you had no idea what on earth '`M-C-x`' should mean, consider taking the *Emacs Tutorial*, which is accessible through the *Help* menu in emacs. (or the '`C-h t`' keyboard shortcut). Beyond any reasonable doubt, emacs is among the most versatile and flexible editors in the universe. Many IFI courses recommend (or presume) the use of emacs, and quite generally, fluent emacs users are better citizens; emacs can be a great tool for any editing task, not just programming. Many people, for example, used emacs to read and write email or to compose and typeset term papers and love letters (often using LaTeX).

2 List Selection

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))`

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 (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 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.

7 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
```

8 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)
```

9 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.

(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 (ppcre:all-matches-as-strings "[^ ]+" 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 for the first exercise.

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

10 Experimenting with Regular Expressions

(a) Grefenstette & Tapanainen (1994), a seminal (albeit somewhat sloppily written) research report on how to tokenize and segment a corpus into sentences, suggest the following rules for the detection of abbreviations:

- a single capital letter, followed by a period, e.g. '`A.`', '`B.`', '`C.`', or '`.`';
- a single letter, followed by a period, followed by one or more pairs of a single letter plus period, e.g. '`U.S.`', '`i.e.`', or '`m.p.h.`';
- a capital letter followed by a sequence of consonants and a period, e.g. '`Mr.`' or '`Assn.`'

(b) Translate the above patterns into regular expressions and determine how many of the tokens in `*corpus*` match either of the three descriptions. Note that for this exercise, we are probably only interested in matches for the whole token, i.e. no matches on only a sub-string of a token. Next, combine all three patterns into a single regular expression and determine the total count of candidate abbreviations.

(c) Why is the third condition above restricted to sequences of consonants only? Try a more liberal pattern and determine experimentally what can go wrong with it, i.e. see whether there are tokens that match the new pattern, even though they actually are not abbreviations.

(d) To search for additional abbreviations in `*corpus*`, print out all tokens that end in a period and do *not* match any of the rules above. Can you spot additional examples that should ideally be recognized as abbreviations? See whether you can further refine the above rules, without adding too many false matches (see below).

11 Submitting Your Results

Please submit your results in email to Stephan ('`oe@ifi.uio.no`') and Johan ('`johanbev@ifi.uio.no`') before midnight on Wednesday, February 4. Ideally, please provide a single text file, including your code and answers 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).