Computational Linguistics (Spring 2008, 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.

1 Bring up the Editor and the LKB Lisp Environment

(a) Login to the Linux server '`login.ifi.uio.no`' through X Windows. To prepare your account for use in this class, you need to perform a one-time configuration step. At the shell command prompt (on '`login.ifi.uio.no`'), execute the following command:

```
~oe/bin/inf2820
```

This command will add a few lines to your personal start-up files '`.bashrc`' and '`.emacs`'. For these settings to take effect, you need to log out one more time (from the session on '`login.ifi.uio.no`' only) 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).

(b) Launch the LKB grammar development and Lisp environment that we will use throughout the semester, by typing (at the shell prompt):

```
lkb &
```

This will lauch 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.

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))`

(d) `(apple (orange) ((pear (lemon))))`

(e) `(apple (orange (pear (lemon))))`

3 List Construction

Show how each of the lists from Exercise 2 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 Multiple Recursion (0 Points)

(a) Write a unary recursive function `flatten()` that takes a list as its argument and loses all embeddings inside of the list, i.e. accumulates all non-list elements of the input in one flat list; e.g.

```
? (flatten '((a) (b ((c)))))
→ (A B C)
```

Note: Although we may not have experienced it so far, it is not unusual for recursive functions to have more than one base case (where the recursion terminates) and call themselves more than once in the recursive branch; use `cond()` in the definition of `flatten()` and note the effects of, e.g.

```
? (append '(a b c) nil)
```