Synopsis Lab 2

This lab is chiefly concerned with writing LISP functions. Remeber that, in FP, we have no sequence of instructions, just function composition; branching is usually done via `COND`; iteration, by recursion.

For the time being, the best approach is to have the LISP code (i.e. functions definitions) stored in a source file and to load it into `gcl` each time something gets modified. First, you should open two terminals: #1 for source editing via `mcedit` (remember pairing parentheses!) and #2 for running the LISP programs in the GCL interpreter. You start the interpreter by typing `gcl` in terminal #2. To stop it, simply type `(quit)` at its prompt.

By convention, the name of a file containing a GCL source code ends in `.lsp`. If you want to load file `f.lsp`, you should type at the `gcl`'s prompt:

```
(load "f.lsp")
```

By default, the code in `gcl` is interpreted, but we can also have it compiled. The compiled version of a LISP function is identical to its non-compiled version from the point of view of name and parameters, but typically works better as the code is optimized[1]. If you have a source file called, let's say, `mylength.lsp` and want to compile all the functions inside, you should start `gcl` and type:

```
(compile-file "mylength.lsp")
```

`gcl` will create a new file called `mylength.o` to store the compiled versions of the functions in `mylength.lsp`. In order to use the compiled versions, you should first load them by typing:

```
(load "mylength.o")
```

LISP comment lines start with a semicolon[2] (i.e., this sign `;`). Multiple line comments are enclosed between `#|` and `|#`.

Browse Lab #2 and #3, including examples in the section indicated. You should learn that:

1. `DEFUN` is used to define new functions: `(DEFUN function_name (p1 ... pm) f1 ... fn)` When the function is called, the parameters `p1,...,pm` are bound to the values passed by the caller function; then, forms `f1,...,fn` are evaluated and the result of the last form's evaluation (i.e., `fn`) is returned; finally, the parameter bindings are dropped. LISP functions can have:

    - fixed number of parameters. E.g.

    ```
    (defun CADDDDR (l)
      (car (cddddr l)))

    (defun myplus (x y) (+ 1 x y))
    ```

[1] Add `(declare (optimize (speed 3)))` after a function's parameter list to ask the compiler to optimize its speed

[2] Use 4 of them in the header, 3 at the beginning of a comment line, 2 if indented inside the code, 1 otherwise

- optional parameters. E.g. (see file `f.lsp`),

  ```
  (DEFUN f (l &optional end)
      (if (NULL end)  (append l '(period))
                      (append l end)))
  ```

  The call `(f '(This is a sentence) '(exclamation mark))` returns `(this is a sentence exclamation mark)`, while the call `(f '(This is a sentence))` returns `(this is a sentence period)`

- variable number of parameters. In the following example, the function `avg` can be called with as many arguments as we want.

  ```
  (DEFUN avg (&rest li)
     (/ (eval (cons '+ li)) (length li)))
  ```

  The call `(avg 1 2 3)` returns `2`, while the call `(avg 1 2 3 4 5)` returns `3`. The `cons` inside the function just adds a `+` to the argument list, then `eval` forces summing up the resulting list. The last form, `(length li)`, returns the number of elements in `li` (its length).

2. LISP makes use of recursion in writing functions. In order to use this mechanism properly, the following issues to be addressed:

   (a) how to decompose the problem into a simpler versions of itself?

   (b) how can the results of these simpler versions be one combine in order to get the result of the original problem?

   (c) which are the base cases in which the problem can be solved without involving recursion?

   (d) which are the conditions in which the base cases appear?

   E.g.: write a function `mylength` which returns the length of the list given as argument (e.g. `(mylength '(a b c))` should return `3`).

   (a) decomposition: if we knew `mylength` of the list's `rest`, we would be able to return the original list's length

   (b) combine simpler versions: `(+ 1 (mylength (REST l)))`

   (c) base cases solvable without involving recursion: just 1 case, namely the empty list which has length `0`

   (d) conditions for the base cases: `(NULL l)`

   The whole function would be (see file `mylength.lsp`):

   ```
   (defun mylength (l)
       (cond ((NULL l) 0)
             (T (+ 1 (mylength (rest l))))))
   ```

   Let us consider now the well known problem of reversing a list: we need a function which, when given, for example, list `(1 2 3 4 5)`, returns the list `(5 4 3 2 1)`. There are more versions of the function which achieves this; they are stored in the file `revall.lsp` and separately in `rev0.lsp` and `rev1.lsp`.

The first solution for list reversing function is `rev0`:

```
(defun rev0 (ls)
  (if (NOT (NULL ls))
      (ap (rev0 (REST ls))
          (LIST (FIRST ls)))))

(defun ap (ls1 ls2)
  (if (NULL ls1) ls2
      (CONS (CAR ls1) (ap (CDR ls1) ls2))))
```

This solution makes use of `ap`, a user-defined version of the system function `append`. Its purpose is to allow us to trace it. Function `rev0` is not tail recursive (according to the definition in Lecture #2), which makes it quite inefficient. To understand why, we will trace it; for this, you should type:

`(trace rev0)`

`(trace ap)`

`(rev0 '(1 2 3 4))`

Indented function names show you nested calls (preceded by the nesting level and the `>` sign). The `<` sign means a return. We used the user-defined function `ap` in order to be able to trace it; it worth noticing how many times this function is called (hence the lack of efficiency of `rev0`). When you want to stop tracing the function `rev0`, you should do

`(untrace rev0)`

If you want a more detailed (and verbose!) view of a function's behavior, you may use:

`(step (rev0 '(1 2)))`

which shows you which form is evaluated at each point.

A version of `rev` which uses an accumulator is given below:

```
(defun rev1 (ls)
  (rev1ac ls NIL))

(defun rev1ac (ls res)
  (cond ((ENDP ls) res)
        (T (rev1ac (REST ls) (CONS (FIRST ls) res)))))
```

See lecture #2 for the advantages of this approach.

Exercises:

1. trace the functions `rev0` and `rev1` in `revall.lsp` and study their behavior

2. compile the functions `rev0` and `rev1` in the corresponding separate files and notice the message about replacing recursion with iterations. Trace the compiled versions of the functions. For testing, you may want to use functions `nelems` and `f`, which build lists of `n` elements in an automatic manner.

3. write a LISP function called `fifth-element` which takes one list as parameter and returns the fifth element of that list (or `NIL` if the list length is lower than 5). E.g.:

   `(fifth-element '(1 a 2 b 3 c))` will return `3`.

   Can you implement the function in 2 different ways?

4. take a look at the functions `exp0` and `exp1` in Lab #3, section 5. Write another LISP function `power` with two numbers, `m` and `n` as arguments and which calculates m^n using the identity:

$$m^n = \begin{cases} 1 & \text{if n} = 0 \\ m^{\frac{n}{2}} * m^{\frac{n}{2}} & \text{if n} > 0 \text{ and n even} \\ m * m^{\frac{n-1}{2}} * m^{\frac{n-1}{2}} & \text{if n} > 0 \text{ and n odd} \end{cases}$$

5. write a LISP function `fib` which takes one integer `n` as argument and computes the `nth` Fibonacci number (see lectures #1 and #2). E.g. `(fib 6)` will return `8`. Implement the function both without and with accumulators.

6. write a LISP function `sumall` which takes one list as argument and sums up all numbers inside the list. You will have to consider 3 versions of this problem:

   (a) the list is flat, i.e. it contains no nested lists. E.g. `(sumall '(1 a 2 b 3 c))` will return `6`.

   (b) the list does contain nested lists, but the function sums up only the numbers on the list's superficial level. E.g. `(sumall '(1 a 2 b (3 c)))` will return `3`.

   (c) the list does contain nested lists and the function sums up the numbers on all the list's nested levels. E.g. `(sumall '(1 a 2 b (3 c)))` will return `6`.

   Implement the functions both without and with accumulators.