Synopsis Lab 1

The easiest way to work at this lab is to open 2 terminals. In terminal #1, you edit the source file. You should use `mcedit` as it helps you properly pair parentheses; this will be very useful. In terminal #2, you run your LISP programs in the GNU Common Lisp (GCL) interpreter. To start it, type:

```
gcl
```

You quit by typing:

```
(quit)
```

If something goes wrong while in `gcl`, you get into an error state. To return from it, repeatedly press `1` until you reach the Top level. You know you got there when the prompt looks like this: `>`

Browse Lab #1 and #2, including examples in the section indicated. You should learn that:

1. LISP is a language for Functional Programming, a programming paradigm which states that any program is a function applied to some arguments. The basic blocks of LISP are **atoms** and **lists**.

2. Atoms are either numerical (e.g., `112`, `3.14`) or strings (e.g., `"f1.lsp"`) or symbolic (e.g., `a`). The latter ones resemble variables as they can have values assigned - "bound". Their names are not case sensitive. Symbolic atoms also include names of primitive functions (e.g. `MEMBER`).

3. A list starts with `(`, ends with `)` and contains atoms and/or some other lists. E.g., `(1 (2) 3)`

4. Symbolic atoms have values bound to them; e.g. `1`, `T` and `NIL` are bound to themselves. Atom `NIL` is also a list as it represents the empty list: `()`.

5. a **form** is something which can be evaluated:

   - a string or numerical atom - they are evaluated to their own value
   - a symbolic atom - evaluated to the value bound to it
   - a list having a function name in the first position and the function arguments in the other positions, e.g., `(f 1 a)` means $f(1, a)$, where `a` should have some value bound to it. First, the arguments are evaluated, then the function is applied to them. Arguments of a form could also be forms, thus allowing function composition.

6. the LISP interpreter executes an infinite READ-EVAL-PRINT loop (REPL in short). By default, the interpreter aims to evaluate every form it sees. The evaluation of one form could be prevented explicitly by placing a quote in front of it (e.g., `(f 1 'a)`). The quote character is a shortcut for `QUOTE`, a function which returns the argument received without evaluating it.

7. arithmetic functions: `+,-,*,/,MIN,MAX`. Most of them are used in prefix style, e.g., for 1+2 we do `(+ 1 2)` (see examples in L01, section 2.3.1)

8. selectors: (see examples in L01, section 2.3.2)

   - `CAR` aka `FIRST`: returns the first element in the list given as argument, e.g.,
     `(CAR '(1 2 3))` returns `1`
   - `CDR` aka `REST` returns the list given as argument with the first element eliminated, e.g.,
     `(CDR '(1 2 3))` returns `(2 3)`
   - `SECOND` returns the second element in the list given as argument, e.g.,
     `(SECOND '(1 2 3))` returns `2`
   - `LAST` returns the list containing the last element in the list given as argument, e.g.,
     `(LAST '(1 2 3))` returns `(3)`
   - `NTH` returns the element in the list given as argument, at the specified position e.g.,
     `(NTH 0 '(1 2 3))` returns `1`
   - `NTHCDR` returns the list containing the elements in the list given as argument, starting from the specified position e.g.,
     `(NTHCDR 1 '(1 2 3))` returns `(2 3)`

9. binding: `SET, SETQ, SETF` (see examples in L01, section 2.3.4). Let's do:

   `(SETQ li '(a b c))`

   `li` will be `(a b c)` and `a` will be unbound. If you do now:

   `(SET (CAR li) 1)`

   then `li` will be `(a b c)` (not modified) and `a` will be 1. But if you do instead:

   `(SETF (CAR li) 1)`

   then `li` will be `(1 b c)` (you modified something inside the list itself) and `a` will be unbound.

10. constructors: (see examples in L01, section 2.3.3)

    - `CONS` adds an element in front of a list, e.g.,
      `(CONS 1 '(2 3))` returns `(1 2 3)`
      For any `x` and `y`, `(x . y)` is the same as `(CONS x y)` (mind the spaces around the dot !)
      When using the dot[1], the result can be seen as a pair of pointers, to the first and the second argument respectively. So, `(1 2)` is the same as `(CONS 1 '(2))` or as `'(1 . (2 . NIL))`; the latter representation is called the dot notation. The same holds for `'(1 . 2)`, which is the same as `(CONS 1 2)`; the result is not a proper list and is called "dotted pair".
    - `APPEND` concatenates the lists given as arguments, e.g.,
      `(APPEND '(10) '(11 12))` returns `(10 11 12)`
      Keep in mind that `APPEND` has a peculiar behavior: it creates copies of its first `n-1` arguments and uses the original copy of its last. E.g.

      ```
      (setq l1 '(a b))
      (setq l2 '(c d))
      (setq l (APPEND l1 l2))
      ```

[1] Technically, the dot `.` is a macro

`l1` will evaluate to `(a b)`, `l2` to `(c d)` and `l` to `(a b c d)`. If we do:

```
(setf (CAR l1) 3)
```

`l1` will evaluate to `(3 b)` and `l` to `(a b c d)` (not changed as it contains a copy of `l1`, not the genuine `l1`). If we do further:

```
(setf (CAR l2) 4)
```

`l2` will evaluate to `(4 d)` and `l` to `(a b 4 d)` (`l` is modified as well because it does not contain a pointer to a copy of `l2` but to `l2` itself).

- `LIST` puts all its arguments into a list, e.g.,
  `(LIST 2 3 5 7)` returns `(2 3 5 7)`
- `REVERSE` reverses the list given as its argument, e.g.,
  `(REVERSE '(2 3 5 7))` returns `(7 5 3 2)`
- `SUBST` replaces every occurrence of its second argument with its first argument on the list given as its third argument, on every nesting level, e.g.,
  `(SUBST 1 'one '(1 (1 2) go))` returns `(1 (1 2) GO)`

11. predicates are functions which return either false (`NIL`) or true (everything different from `NIL`, e.g. `T`). Usually, their names end in `P`:
    - `NUMBERP` checks if its argument is a number;
    - `SYMBOLP` checks if its argument is a symbolic atom;
    - `LISTP` checks if its argument is a list;
    - `ZEROP` checks if its argument is `0`;
    - `ENDP` (aka `NULL`) tests if it argument is an empty list

    (see examples in L01, section 2.3.5)

12. `ATOM` is a predicate which tests if its argument is an atom of any kind

13. `EQ` tests for equality (the arguments are precisely the same: they have the same place in memory); `EQUAL` tests for isomorphism (the arguments have the same structure, even if are not in the same place in memory). The difference matters for lists, as there might exist 2 different copies of `(a b)`; on the other hand, two atoms are always EQ (and EQUAL) since there is just one copy, in the atom memory, of, let's say, number 1. See examples in L01, section 2.3.5, L02, section 2.1.2; try also:

    ```
    (SETF (SECOND a) 'k)
    ```

    and note that `a` evaluates to `((a b) k (a b))` and `b` evaluates to `(k (a b))`

14. `MEMBER` tests whether its first argument is member of (i.e., can be found into) the list given as its second argument. It returns either `NIL` or a sublist of argument #2 starting with the first occurrence of the first argument; the test is done using `EQ`. E.g.,

    `(MEMBER 5 '(1 (5) 5 3 5 1))` returns `(5 3 5 1)`

    `(MEMBER '((5)) '(1 (5) 5 3 5 1))` returns `NIL`

    `MEMBER` employs `EQ`, but this can be changed using `:TEST` (see examples in L01, section 2.3.5)

15. `PRINT` prints something on the screen (see examples in L01, section 2.3.6)
16. `READ` reads something from the keyboard (see examples in L01, section 2.3.6)
17. `EVAL` forces one more evaluation. See examples in L02, section 2.2.1 and L02, section 2.2.2.
18. `OR` evaluates its parameters from left to right until the first form returns something which is not `NIL`; this is the result returned by `OR`. If all forms return `NIL`, the result returned by `OR` is also `NIL`. E.g.

    ```
    (OR (setf x 'black) (setf y 'white))
    ```

    binds `x` to `black` and leaves `y` unbound
19. `AND` evaluates parameters from left to right until one form returns `NIL`; this is the result of `AND`. If all forms return something not `NIL`, the result returned by `AND` is the result of the last evaluated form.
20. `COND` does branching:

    ```
    (COND (v1 f11 ...  f1m)
          (v2 f21 ...  f2n)
          ...
          (vp fp1 ..  fpq))
    ```

    `v1`, then `v2` etc. are evaluated until the first one which is not `NIL`; then, the forms `fij` inside that branch of `COND` are evaluated and the result of `COND` is the result of evaluating the last form `fij`. If all `vi`'s are `NIL`, then `COND` returns `NIL`.
21. `IF` is a particular form of `COND`: `(IF C FT FF)` evaluates form `C` and, if it is `T`, returns the result of evaluating `FT`, otherwise the result of evaluating `FF`

Exercises:

- L01 3.2: indicate the type of elements at a,b,e,f
- L01 3.3: evaluate the first 3 forms
- L01 3.4: write the LISP form which computes the first expression
- L01 3.5: evaluate:

  ```
  (member a a)
  (member 'nice '(what a nice day))
  (append nil nil nil)
  (reverse (reverse p))
  ```
- L01 3.6: write forms for extracting atom `alfa` from the given lists
- L01 3.7: indicate the forms which give an error if evaluated and explain why
- L02 3.1: write the lists at 3.1.a and b in the dot notation .
- L02 3.3: draw the representation of the lists at 3.1.a,b,c,g
- L02 3.4: evaluate the first 8 forms (on the first 4 lines) at 3.4
- L02 3.7: evaluate the forms in 3.7