Synopsis Lab 3

Examples for this lab are included in this Synopsis or in Lab #5 or #6. You should learn that:

1. A lambda expression is a **nameless function**. It is represented in LISP as a list having the word `lambda` on the first position, a list of parameters (as any other function) on the second position, followed by a number of forms which represent the expression's body (i.e., what the lambda expression actually does). E.g., `(LAMBDA (x y) (+ x y))` is a lambda expression with two arguments (`x` and `y`) which computes their sum (`(+ x y)`).

   Lambda expression are meant to be *applied* on arguments passed to them. As an example, `((LAMBDA (x y) (+ x y)) 1 2)` returns `3` (`x` is bound to `1`, `y` to `2` and their sum is computed).

2. Lambda expressions are used to store a function's body; this makes sense since data and functions' code are regarded in the very same way in LISP, namely as lists.What `DEFUN` actually does is creating a link between the name of the function and the lambda expression describing its body. One can retrieve the body of the function `f` by calling `(SYMBOL-FUNCTION 'f)`. E.g.

   ```
   >(DEFUN foo (x y) (+  x y))  ;define a function
   FOO
   >(foo 3 4) ;call the function
   7
   >(SYMBOL-FUNCTION 'foo) ;retrieve its body
   (LAMBDA-BLOCK FOO (X Y) (+ X Y))
   >(SETF (SYMBOL-FUNCTION 'foo) '(LAMBDA (x y) (* x y))) ;replace body (redefine)
   (LAMBDA (X Y) (* X Y))
   >(foo 3 4) ; call the function with the redefined body
   12
   ```

3. In order to apply a function/lambda expression bound to a symbol (e.g. if we need to send a lambda expression as a parameter to another function), we can use either `FUNCALL` or `APPLY`:

   - `(FUNCALL f a1 ... an)` evaluates its first parameter `f` and, if the result is a function or a lambda expression, it applies it on the remaining arguments `a1 ... an` . A typical use is the situation when we want to send functions as parameters to some other functions or to call a function whose body has been built by a LISP code.
   - `(APPLY f l)` evaluates its first parameter `f` and, if the result is a function or a lambda expression, it applies it on the arguments contained inside the list `l`. A typical use is the situation when we want to build in a dynamic manner, from inside a piece of code, the list of parameters which are to be sent to a specific function. E.g.

```
>(DEFUN foo (x) (+ x 10)) ;define a function
FOO
>(SETQ foo #'(LAMBDA (x) (* x 10))) ;bind a lambda expression to foo
(LAMBDA-CLOSURE () () () (X) (* X 10))
>(foo 5) ;standard function call
15
>(FUNCALL foo 5) ;evaluate foo, then apply the returned lambda expression
50
>(FUNCALL #'foo 5) ;foo is not evaluated, just called
15
>(APPLY foo '(5)) ;evaluate foo, then apply the returned lambda expression
                  ;on the arguments stored in the list
50
>(APPLY #'foo '(5)) ;apply foo on the arguments stored in the list,
                    ;without previously retrieving the returned value
15
>(FUNCALL (LIST 'LAMBDA '(x y) '(+ 1 x y)) 2 3) ;build a function on the
                                                ;fly and call it
6
>(SETF a 1 b 3 c 2)
2
>(APPLY #'* (LIST a b c)) ;build argument list on the fly
6
>(DEFUN bar (fun-par) (FUNCALL fun-par 'alpha))
BAR
>(bar #'LIST) ;pass a function name as an argument to another function
(ALPHA)
>(bar #'ATOM)
T
```

If we want to prevent a lambda expression from being applied, we should use `#'` in front of it. Remeber that `'` (a shortcut for `QUOTE`) prevents an atom from being evaluated. Analogously, `#'` (shortcut for `FUNCTION`[1]), gets the function body of `i`. Hence, if you do `(defun i (x) x)` then `(setq i 3)`, `'i` gives `I`, while `#'i` gives `(SYSTEM:LAMBDA-BLOCK I (X) X)`.

4. The rules for variable binding and scoping are the same for lambda expressions and functions. To understand them, let us conside the following example (see file `link.lsp`):

```
(DEFUN f ()
  (LET*  ((x 2)
         (foo #'(LAMBDA () (PRINT x)))
         (bar #'(LAMBDA (x) (PRINT x) (FUNCALL foo))))
        (FUNCALL bar 1))
  (print x))
```

If you do `(SETQ x 3)`, then call `f`, you will get `1,2,3,3`. The explanation is as follows.

First, let us take a look at the `LET*` form. Every local variable in `LET*` (`x`, `foo` and `bar`) gets bound temporarilly to the value returned by its associated form (`x` is bound to `2`, `foo` to

[1] `QUOTE` and `FUNCTION` are in fact macros. Their actual parameters replace the formal ones *without* prior evaluation.

(LAMBDA () (PRINT x)) etc.); bindings happen sequentially. Then, the forms in the body of LET* (just one in our case) are evaluated in sequence and the result of the last one is returned.

But which is the value of x in each of the form in f's body? LISP's default strategy for answering this question is called "lexical binding". It asserts that the value of a variable v with no binding in a block B is the value or x in the innermost scoping block which encloses B. A variable with no binding in a block is called "free" in that block.

Because x is bound in bar (it appears in the parameter set of lambda), its value is the one sent by the call (FUNCALL bar 1). The first PRINT will display 1. Then foo is called and prints the value of x in LET*, which is 2 (x is free in foo, so the value in the innermost lexical block encompassing it, namely LET*, is displayed). The value displayed by the last print is 3, the one set by SETQ (the lexical block encompassing this one is the global lexical block, we are outside the LET*). The last 3 is the value returned by f.

5. MAPCAR f l1 ... ln takes as first argument a function/lambda expression f of n arguments and calls it on tuples built from corresponding elements in the lists l1,...,ln. The result is a list containing the results of these successive calls. E.g.

```
>(MAPCAR #'ODDP '(1 2 3))
(T NIL T)

>(MAPCAR #'CONS '(1 2 3) '(1 4 9))
((1 . 1) (2 . 4) (3 . 9))

>(mapcar #'(LAMBDA (x y z) (+ x y z)) '(1 2 3) '(4 5 6) '(7 8 9))
(12 15 18)
```

6. an association list is a list consisting of CONS cells of type (key . value). The search functions for association lists are ASSOC and RASSOC. The default equality test is EQ, but this can be changed via :TEST. Search is done in linear time.

Function (ASSOC key alist) returns the first pair in the association list alist which has its CAR the same as key, or NIL if no such a pair is found.

Function (RASSOC val alist) returns the first pair in the association list alist which has its CDR the same as val. E.g.

```
>(SETQ lsq '((1 . 1) (2 . 4) (3 . 9)))
((1 . 1) (2 . 4) (3 . 9))
> (ASSOC 3 lsq)
(3 . 9)
>(RASSOC 4 lsq)
(2 . 4)
```

Exercises:

1. Evaluate expressions in L05, 3.1, up to (APPLY 'CAR '((a b))).
2. Given the the function (see our-remove-if.lsp):

```
(DEFUN our-remove-if (pred l)
  (MAPCAN #'(LAMBDA (x) (IF (NOT (FUNCALL pred x)) (LIST x))) l))
```

What does the following call return:

`(our-remove-if #'(LAMBDA (x) (EQUAL x 'a)) '(r a d i o g a g a))`

3. Let us assume we do:

```
(defun i (x) x)
(setq i 3)
```

Evaluate the forms below and explain the results:

```
i
'i
#'i
(i i)
(i #'i)
```

4. You are given the following list of CONS cells representing countries and their capitals:

```
(SETQ atlas '((France . Paris) (Romania . Bucharest) (Germany . Bonn)))
```

and the following function which lists all capitals (see the `map.lsp` file):

```
(DEFUN selcap (li)
  (MAPCAR #'CDR li))
```

Using `MAPCAR`, write the following functions:

(a) `(selcountries (li))` which lists all countries in the atlas `li`.
Example: `(selcountries atlas)` should return `(FRANCE ROMANIA GERMANY)`

(b) `(selrom (li))` which returns the list of all cities mentioned as capitals of Romania in `li`.
Example: `(selrom atlas)` should return `(BUCHAREST)`
Can you come up with two solutions?

(c) `(moveGermCap (li))` which moves the capital of Germany from Bonn to Berlin
Example: after doing the call `(moveGermCap atlas)`, the atom `atlas` should evaluate to
`((FRANCE . PARIS) (ROMANIA . BUCHAREST) (GERMANY . BERLIN))`

Now test the functions `ASSOC, RASSOC` on querying the associations list above.

5. (a) Write a function `(dbl (e))` which returns `2*e` if the `e` is a number and `e` otherwise.
Example: `(dbl 1)` should return `2`, but `(dbl "one")` should return `"one"`.

(b) Write a function `(proc-rec (process li))` which applies the one-argument function `process` over every element in a (possibly nested) list `li` and returns the list of results.
Example: `(proc-rec #'(LAMBDA (x) (+ 1 x)) '(1 (2) 3))` should return `(2 (3) 4)`

(c) Write a function `(f (li))` which doubles every number in a list; all other elements should be left unchanged. Function `f` will actually be a wrapper for `proc-rec`, which calls it and passes `dbl` as parameter # 1 and `li` as parameter # 2. Example:
`(f '(1 2 (3 a) 4 b))` should return `(2 4 (6 A) 8 B)`