LECTURE 10 - Evoluated definitional mechanisms

Contents

1. The EVAL form

Applying the EVAL form is equivalent to calling the Lisp evaluator. The syntax of the function is

EVAL subr 1 (f): e

The effect consists in evaluating the form and returning the result of the evaluation. The form is actually evaluated in two stages: first it is evaluated as an argument of EVAL and then the result of this evaluation is again evaluated as an effect of the application of the EVAL function. E.g:

- (SETQ X '((CAR Y) (CDR Y) (CADR Y)))
- (SETQ Y '(A B C))
- (CAR X) is evaluated at (CAR Y)
- (EVAL (CAR X)) will produce A

The mechanism is similar to what indirectization means by pointers in imperative languages.

- (SETQ L '(1 2 3))
- (SETQ P '(CAR L))
- P is evaluated at (CAR L)
- (EVAL P) will produce 1

- (SET B 'X)
- (SETQ A 'B)
- (EVAL A) is evaluated at X
- (SETQ L '(+ 1 2 3))
- L is evaluated at (+ 1 2 3)
- (EVAL L) is evaluated at 6

Remark. It should be noted here that the Lisp system does not evaluate the first element of a form, but only applies it.

What we want to emphasize is that the association (SETQ Q 'CAR) does not allow the call in the form (Q '(A B C)), such a call signaling error in the sense that the LISP evaluator does not find any function with the name Q. On the other hand, even with the help of the EVAL function we cannot solve the problem:

- (SETQ Q 'CAR)
- (SETQ P 'Q)
- (EVAL P) will produce CAR
- ((EVAL P) '(A B C)) will produce an error message: "Bad function when ..."

The above error message appears because Lisp does not evaluate its first argument in a form. Be careful, however, not to confuse this statement with the implicit mechanism of action of the Lisp evaluation, ie any list is implicitly interpreted as a form, trying to evaluate and considering the first argument as the function to be applied. So a list is always evaluated if this is not explicitly stopped (by QUOTE), instead the first argument of any list is never evaluated!

2. Functional forms

In Lisp, the arguments of the functions are usually evaluated before the evaluation of the calling functions. We have already seen the QUOTE function which has the effect of preventing the evaluation of the argument to which it refers. A similar role has the function FUNCTION, which has the effect of treating that argument as a function name.

Argument		
Function f	**#'f**	(function f)
Symbol x	**'x**	(quote x)

	Function f
Standard	#'f
CLisp	'f
GCLisp, Emacs Lisp, other dialects	'f

	Lambda expression
Standard	#'(lambda…)
CLisp	(lambda…)
GCLisp, Emacs Lisp, other dialects	'(lambda…)

3. APPLY and FUNCALL functions

There are situations in which the form of the function is not known, its expression having to be determined dynamically. We should have something like that

(EXPR_FUNC p1 ... pn)

Because EXPR_FUNC must eventually generate a function it is a so-called functional form. The EXPR_FUNC form is evaluated until a function or, in general, an expression that can be applied to the parameters is obtained.

In such a situation, the evaluation of the parameters is postponed until the moment of reducing the functional form EXPR_FUNC to the actual function F. The parameters will be evaluated only if F is of type subr, lsubr or expr. So the evaluation of the above form goes through the stages:

(i) reduction of the EXPR_FUNC form to F (possibly a LAMBDA expression or macrodefinition) and substitution of EXPR_FUNC by F in the form to be evaluated;
(ii) form evaluation (F p1 ... pn).

However, there are situations in which the number of parameters must be set dynamically, so the dynamically determined function must accept a variable number of parameters. There is therefore a need for a way to allow the application of a function on a set of parameters that can be synthesized dynamically. This is provided by the APPLY and FUNCALL functions.

APPLY lsubr 2 (ff lp): e

The APPLY function allows you to apply a function to parameters provided as a list. In the above description, ff is a functional form and lp is a form reducible by evaluation to a list of effective parameters (p1 p2 ... pn).

- (APPLY #'CONS '(A B)) produces (A . B)
- (APPLY #'MAX '(1 2 3)) produces 3
- (APPLY #'+ '(1 2 3)) produces 6

FUNCALL lsubr 1- (f ... f): e

FUNCALL is a variant of the APPLY function that allows the application of a function (or expression) resulting from the evaluation of a functional form ff on a fixed number of parameters.

- (FUNCALL #'CONS 'A 'B) produces (A . B)
- (FUNCALL #'MAX 1 2 3) produces 3

- (SETQ P 'CAR)
- ((EVAL P) '(A B C))   incorrect

The above example will become a correct form if FUNCALL or APPLY is applied:

- (FUNCALL (EVAL P) '(A B C)) = A
- (APPLY (EVAL P) '((A B C))) = A

This is because (EVAL P) is no longer the first argument in the form to be evaluated, so it will be evaluated at CAR, and then FUNCALL (or, as the case may be, APPLY) will apply the function designated by the first argument on the list (A B C).

- (DEFUN F (L)   ; returns T if the argument is a numeric atom

    (COND

        ((NUMBERP L) T)

        (T T)

)

)

- (FUNCALL #'F 2) = T
- (FUNCALL #'F 'A) = NIL
- (FUNCALL #'F '(1 2)) = NIL

The difference between APPLY and FUNCALL is that when defining with APPLY the number of arguments can be ignored.

- (DEFUN MYOR (L)   ; returns T if the argument is a list in which the

    (COND   ; at least one item is evaluated at≠ NIL

        ((ATOM L) T)

        ((CAR L) T)

        (T (MYOR (CDR L)))

)

)
- (FUNCALL #'MYOR 2) = NIL
- (FUNCALL #'MYOR '(T NIL)) = T
- (FUNCALL #'MYOR '(NIL NIL)) = NIL

4. LAMBDA expressions

In situations where (i) a one-time function is far too simple to be defined, or (ii) the function to be applied must be synthesized dynamically (it is therefore impossible to define it statically by DEFUN), it may be use a particular functional form called lambda expression. A lambda expression is a list of form

(LAMBDA l f1 f2 ... fm)

which defines an anonymous function usable only locally, a function that has the definition and the call concentrated in the same point of the program that uses them, l being the list of parameters and f1 ... fm representing the body of the function. The list of arguments l can be given explicitly (fixed number of parameters) or parametric (list l - this means a variable number of parameters). Arguments are evaluated on call (expr and lexpr). If the arguments are not to be evaluated on call, the QLAMBDA form must be used.

Such a form of LAMBDA is commonly used as follows:

((LAMBDA l f1 f2 ... fm) par1 par2 ... parn)

Example:

(FUNCALL #'(LAMBDA (L) (CAR L)) '(A B C)) = A

The use of this kind of definition in the presence of recursion becomes impossible, being anonymous functions. Appearing the same definition in two or more places already means redefining, so we can't actually make the recursive call.

5. The LABELS form

A special form to locally bind functions is the form LABELS.

(LABELS bindings body)

where:

bindings   list containing function definitions (without the DEFUN particle)
body   program code in which definitions above are effective

Example

Ex1. The evaluation

```
(labels ((fct(l)
              (cdr l)
         )
       )
       (fct '(1 2))
)
```

will produce (2).

Ex2.

```
(labels ((temp (n)
              (cond
                   ((= n 0) 0)
                   (t (+ 2 (temp (- n 1))))
              )
        )
       )
       (temp 3)
)
```

will produce 6.

Ex3. Write a function that receives as a parameter a list of lists consisting of atoms, and returns T if all lists contain numeric atoms and NIL otherwise.

(test '((1 2) (3 4))) = T
(test '((1 2) (a 4))) = NIL
(test '((1 (2)) (a 4))) = NIL

Solution:

```
(DEFUN TEST (L)
      (COND
            ((NULL L) T)
            ((LABELS     ((TEST1 (L)
                              (COND
                                    ((NULL L) T)
                                    ((NUMBERP (CAR L)) (TEST1 (CDR L)))
                                    (T NIL)
                               )
                              )
                              )
                (TEST1 (CAR L))
            ) (TEST (CDR L)))
            (T NIL)
      )
)
```

6. Using LAMBDA expressions to avoid repeated calls

Ex1. Consider the following function definition

```
(defun g(l)
      (cond
             ((null l) nil)
             (t (cons (car (f l)) (cadr (f l)))))
      )
)
```

The solution to avoid the duplicate call (f l) is to use an anonymous function locally, which can be called with the actual parameter (f l).

Variant 1

```
(defun g(l)
      (cond
             ((null l) nil)
             (t ((lambda (v)
                    (cons (car v) (cadr v))
                 )
                (f l)
                )
             )
      )
)
```

Variant 2

```
(defun g(l)
      ((lambda (v)
             (cond
                    ((null l) nil)
                    (t (cons (car v) (cadr v)))
             )
      )
      (f l)
      )
)
```

Ex2. Consider the definition of the function that generates the list of subsets of a set represented as a list.

```
(defun subm(l)
      (cond
             ((null l) (list nil))
             (t (append (subm (cdr l)) (insPrimaPoz (car l) (subm (cdr l))))))
      )
)
```

As you can see, the call **(subm (cdr l))** is repeated. To avoid the repeated call, we will use a LAMBDA expression.

A possible solution is the following:

```
(defun subm(l)
      (cond
             ((null l) (list nil))
             (t ((lambda (s)
                    (append s (insPrimaPoz (car l) s))
                  )
                  (subm (cdr l))
                  )
             )
      )
)
```

7. Generators. Functional arguments

Consider the example of a function that receives a list of lists consisting of atoms and returns T if all lists contain numeric atoms and NIL otherwise. The TEST function goes through the list of arguments, the actual test being performed by the TEST1 function:

```
(DEFUN TEST (L)
      (COND
            ((NULL L) T)
            ((TEST1 (CAR L)) (TEST (CDR L)))
            (T NIL)
      )
)



(DEFUN TEST1 (L)
      (COND
            ((NULL L) T)
            ((NUMBERP (CAR L)) (TEST1 (CDR L)))
            (T NIL)
      )
)
```

Note that the TEST and TEST1 functions above have the same structure, according to the pattern

```
(DEFUN F (L)
    (COND
        ((NULL L) T)
        ((F1 (CAR L)) (F (CDR L)))
        (T NIL)
    )
)
```

with F1 = TEST1 in case of F = TEST and F1 = NUMBERP in case of F = TEST1. Such a structure is often used, its working "recipe" being: the elements of a list L are transmitted in turn (in the order of their appearance in the list) to a function F (in the case above F1) that will process them. If F returns NIL the action ends, the end result being NIL. Otherwise, scrolling until all the elements are exhausted, in which case the final result is T. As this recipe can be applied to any function F that performs the processing in a certain way of all the elements of a list, the idea of writing a generic function appears GEN that respects the "recipe", having two parameters: the function that will perform the processing and the list on which to act.

The terminology adopted for such (or similar) functions is that of **generators**. It is not necessary for a generator to return T or NIL. Depending on the implementation, generators can be designed to return, for example, the list of all non-NIL results of F collected until the moment of cessation of generator action (L is empty or F returns NIL). Also, other variants do not impose (CAR L) as the only parameter for F.

```
(DEFUN GEN (F L)
    (COND
        ((NULL L) T)
        (T (FUNCALL F (CAR L)) (GEN F (CDR L)))
    )
)
```

Consider for example the function (LIN L) which has a list of values as a parameter and checks if all sublists of the parameter list are linear (returns T if yes, NIL otherwise).

A brief analysis shows the need for a generator function of the form

```
(DEFUN GEN (F L)
    (COND
        ((NULL L) T)
        (T (AND (FUNCALL F (CAR L)) (GEN F (CDR L)))
    )
)
```

Once the generator has been defined, the function (LIN L) can be defined as follows:

```
(DEFUN LIN (L)
    (GEN #'(LAMBDA (L)
            (GEN #'NUMBERP L)
        )
        L
    )
)
```

8. MAP functions

MAP functions are similar to generators except that scrolling through the list (s) given as a parameter is performed entirely regardless of the value returned by the functional parameter. The role of MAP functions is to apply a function repeatedly to items (or successive sublists) of given lists as arguments.

MAPCAR lsubr 2- (f l1 ... ln): l

The MAPCAR function is a global application function. The n-ary function f is applied in turn to the elements of the argument lists, the results being collected in a returned list as the value of the MAPCAR function call. The evaluation ends at the end of the shortest list. E.g

- (MAPCAR #'CAR '((A B C) (X Y Z))) is evaluated at (A X)
- (MAPCAR #'EQUAL '(A (B C) D) '(Q (B C) D X)) is evaluated at (NIL T T)
- (MAPCAR #'LIST '(A B C)) is evaluated at ((A) (B) (C))
- (MAPCAR #'LIST '(A B C) '(1 2)) is evaluated at ((A 1) (B 2))
- (MAPCAR #'+ '(1 2 3) '(4 5 6)) is evaluated at (5 7 9)

Applying the LIST function is possible regardless of the number of argument lists because LIST is a function with a variable number of arguments.

Let's take another example: defining a MODIF function that modifies a given list as a parameter as follows: non-numeric atoms remain unchanged and numeric atoms double their value; the change must be made at all levels:

```
(DEFUN MODIF (L)
    (COND
        ((NUMBERP L) (* 2 L)); determineoperation on digital atoms
        ((ATOM L) L); determineoperation on non-numerical atoms
        (T (MAPCAR #'MODIF L)); represents the browsing strategy
    )
)
```

According to the same model, let us construct an LGM function that determines the *length of the longest sublist in a given list L* (if the list consists only of atoms then the required length is exactly that of the list L). The description of the algorithm can be expressed as follows:

1. the LGM value is the maximum between the length of the list L and the maximum of the values of the same nature calculated by applying the LGM for each element of the list L separately;
2. LGM (atom) = 0.

```
(DEFUN LGM (L)
    (COND
        ((ATOM L) 0)
        (T (MAX (LENGTH L) (APPLY #'MAX (MAPCAR #'LGM L))))
    )
)
```

Applying the ATOM and LENGTH functions actually calculates the lengths, (MAPCAR #'LGM L) actually completing the entire list. The call (MAPCAR #'LGM L) provides a list of lengths. Since we need to get the most out of them, we will need to apply the MAX function to the items in this list. For this we use APPLY.

MAPLIST lsubr 2- (f l1 ... ln): l

The MAPLIST function applies the n-ary function to all lists, then to all their CDRs, then to all their CDRs (to the initial CDDRs), and so on (to all successive sublists) until one of the sublists becomes NIL. The list of successive results obtained is returned. Example:

- (MAPLIST #'APPEND '(A B C) '(1 2 3)) produces ((A B C 1 2 3) (B C 2 3) (C 3))

- (MAPLIST #'(LAMBDA (X) X) '(A B C)) produces ((A B C) (B C) (C))

- (SETF TEMP '(1 2 7 4 6 5)) followed by

  (MAPLIST #'(LAMBDA (XL YL) (< (CAR XL) (CAR YL)))

      TEMP (CDR TEMP)

) will produce the list (T T NIL T NIL)

Compared to the examples given at MAPCAR, here we get:

- (MAPLIST #'CAR '((A B C) (X Y Z))) is evaluated at ((A BC) (X Y Z))

- (MAPLIST #'LIST '(A B C) '(1 2)) is evaluated at (((A B C) (1 2)) ((B C) (2)))

- (MAPLIST #'LIST '(A B C)) is evaluated at (((A B C)) ((B C)) ((C)))

- (MAPLIST #'EQUAL '(A (B C) D) '(Q (B C) D X)) is evaluated at (NIL NIL NIL)

- (MAPLIST #'+ '(1 2 3) '(4 5 6)) will produce the error message: "+: wrong type argument: (1 2 3); a NUMBER was expected ".

MAPCAN lsubr 2- (f l1 ... ln): l

The MAPCAN function is similar to MAPCAR with the difference that the results obtained are grouped with NCONC in a list that is returned as a result:

- (MAPCAN #'FIRST '((A B C) (X Y Z))) is evaluated at NIL because NCONC requires lists, and as such (NCON 'A 'X) is NIL
- (MAPCAN #'LIST '(A B C) '(1 2)) is evaluated at (A 1 B 2)
- (MAPCAN #'LIST '(A B C)) is evaluated at (A B C)
- (MAPCAN #'EQUAL '(A (B C) D) '(Q (B C) D X)) is evaluated at NIL
- (MAPCAN #'+ '(1 2 3) '(4 5 6)) is evaluated at NIL

MAPCON lsubr 2- (f l1 ... ln): l

The MAPCON function is similar to MAPLIST with the difference that the results obtained are grouped with NCONC in a list that is returned as a result:

- (MAPCON #'FIRST '((A B C) (X Y Z))) provides (A B C X Y Z)
- (MAPCON #'LIST '(A B C) '(1 2)) provides ((A B C) (1 2) (B C) (2))
- (MAPCON #'LIST '(A B C)) provides ((A B C) (B C) (C))
- (MAPCON #'EQUAL '(A (B C) D) '(Q (B C) D X)) provides NIL
- (MAPCON #'+ '(1 2 3) '(4 5 6)) provides error message: "+: wrong type argument: (1 2 3); a NUMBER was expected ".

- (DEFUN G (L)

    (MAPCON 'LIST L)

)
- (G '(1 2 3)) = ((1 2 3) (2 3) (3))

MAPC lsubr 2- (f l1 ... ln): l

The MAPC function is similar to MAPCAR with the difference that it returns the list list l1. As such, when applying this function is important the side effect of evaluating the n-ary function f on the elements of the lists the parameters l1, …, ln are evaluated to.

Remark. This function will NOT be used if the purpose is to return a result (numerical or otherwise) that is obtained by composing in a certain way the results obtained at each level of the list, as this implies the use of procedural facilities (use of SET), and this cancels the advantage of using MAP functions. Here is an example where using MAPC is essential:

```
(DEFUN MYSET (VARLIST VALLIST)
        (MAPC #'SET VARLIST VALLIST)
)
```

The effect is to bind each variable in the VARLIST variable list to the corresponding value in the VALLIST value list:

```
(MYSET '(A B C) '(1 2 3))
```

MAPL lsubr 2- (f l1 ... ln): l

The MAPL function is similar to MAPLIST with the difference that it returns the list l1 as a result. All the remarks made above are valid here as well.