LECTURE 8. Basic functions and predicates in Lisp

Contents

1. Other list constructors

LIST lsubr 0,1, ... (... e ...): l

LIST is a function with a variable number of S-expression parameters. Returns the list of argument values to the surface.

- (LIST ‘(a b) ‘c) = ((a b) c)
- (LIST ‘a) = (CONS ‘a NIL) = (a)
- (LIST ‘(a b) ‘(c d)) = ((a b) (c d))
- (LIST ‘(a b c) ‘()) = ((a b c) NIL) = (LIST ‘(a b c) ())

It is indicated for constructing lists. Suppose we start from the symbols X, Y and Z which have as values A, B and C respectively and that we want the list of values of the symbols X, Y and Z. Then,

• (X Y Z)	is not correct because X will be interpreted as function;
• ‘(X Y Z)	it is not correct because it thus prevents evaluation;
• (LIST X Y Z)	correct and provides the list (A B C).

APPEND lsubr 0,1, ... (... l ...): l

APPEND is a function with variable number of parameter lists (the last can be an object of any type). Copies the list structure of each argument, replacing the CDR of each last CONS with the argument on the right. Returns the resulting list. Note that all atomic parameters except the last parameter are ignored.

- (APPEND ‘(A B) ‘(C D)) = (A B C D)
- (APPEND ‘A) = A

- (APPEND ‘(A B) ‘C) = (A B . C)
- (APPEND ‘(A B C) ‘()) = (A B C)
- (APPEND ‘A ‘B ‘(C D) ‘E ‘F) = (C D . F)

Remarks

- the APPEND function is strongly memory consuming; the first argument list is copied before being linked to the next;
- due to the fact that the last argument of the APPEND function is not copied, it is worth noting what happens when the last argument of the function is destructively modified (for example with SETF):

• (SETQ X ‘(A))	X is evaluated at (A)
• (SETQ Y ‘(B))	Y is evaluated at (B)
• (SETF Z (APPEND X Y))	Z is evaluated at (A B)
• (SETF (CAR X) ‘D)	X is evaluated at (D)
• (SETF (CAR Y) ‘E)	Y is evaluated at (E) but Z is evaluated at (A E)

REVERSE subr 1 (l): l

The REVERSE function returns the inverse of the list received as a parameter. The reversal takes place only at the superficial level:

- (REVERSE ‘(1 2 (3 4))) = ((3 4) 2 1)
- (REVERSE ‘((A B) (C D) (E F))) = ((E F) (C D) (A B))
- (REVERSE ‘((A B C))) = ((A B C))
- (REVERSE ‘(A B C)) = (C B A)
- (REVERSE ‘(A)) = (A)
- (REVERSE NIL) = NIL

Note that REVERSE has no destructive effect:

- (SETQ L ´(A B)) L is evaluated at (A B)
- (REVERSE L) = (B A) L is evaluated at (A B)
- (SETQ L (REVERSE L)) L is assessed at (B A)

LENGTH subr 1 (l): n

The LENGTH function returns the number of elements at superficial level:

- (LENGTH NIL) = 0
- (LENGTH ´(A B)) = 2
- (LENGTH ´(A ((B C) D) E)) = 3

2. Predicates

We mention that the Lisp language has the special atoms NIL, with the meaning of false, and T, with the meaning of true. As in other languages, functions that return a logical value will return only NIL or T. However, it is accepted that any value other than NIL has the meaning of true.

ATOM subr 1 (e): T, NIL

Returns T if the argument is an atom and NIL otherwise.

- (ATOM 'A) = T;
- (ATOM A) = depends on what A is evaluated for;
- (ATOM '(A B C)) = NIL;
- (ATOM NIL) = T;

LISTP subr 1 (e): T, NIL

Returns T if the argument is a list and NIL otherwise.

- (LISTP 'A) = NIL;
- (LISTP A) = T if A is evaluated on a list;
- (LISTP '(A B C)) = T;
- (LISTP NIL) = T;

EQ subr 2 (e e): T, NIL

EQL subr 2 (e e): T, NIL

EQUAL subr 2 (e e): T, NIL

EQ returns T if the arguments are the same identical object and NIL otherwise.

EQUAL returns T if the values of the arguments are equivalent S-expressions (ie if the two S-expressions have the same structure - so the inefficiency of its implementation compared to EQ is clear).

EQL returns T if the arguments are EQ or if they are the same non-structured values (that is, the same numeric values for numbers of the same type or the character values). Since it includes the EQ operator and can be used also on non-structured values, is the most important and most commonly used operator, and almost all the primitive functions that require an equality comparison, like MEMBER, use by default this operator.

- (SET ‘X ‘(A B))
- (SETQ Y ‘(A B))
- (EQ X Y) = NIL
- (EQUAL X Y) = T
- (SET ‘X ‘(A B))
- (SETQ Y X)
- (EQ X Y) = T
- (EQUAL X Y) = T

It is obvious that any two S-expressions that are EQ are always EQUAL.

These results occur due to the unique way in which atoms are represented, as opposed to lists, which are constructed.

e1 e2	EQ	EQL	EQUAL
‘(A B) ‘(A B)	NIL	NIL	T
‘(A B) ‘(A (B))	NIL	NIL	NIL
3 3.0	NIL	NIL	T
3.0 3.0	NIL	T	T
8 8	T	T	T
‘A ‘A	T	T	T

3. Predicates for lists

NULL subr 1 (e): T, NIL

Returns T if the argument is evaluated to an empty list or null atom and NIL otherwise.

MEMBER subr 2 (e l): l

It is used to check the inclusion of an S-expression in a list. Returns from the list to which the second argument evaluates the sublist that begins with the first occurrence of the first argument. If the first argument does not appear in the list, NIL is returned. The equality of S-expressions is checked with EQL. The verification is done only at a superficial level.

- (SETQ X ‘(A B (C D) E))
- (MEMBER ‘C X) = NIL
- (MEMBER ‘B X) = (B (C D) E)
- (MEMBER ‘E X) - (E)
- (MEMBER ‘(C D) X) = NIL
- (MEMBER ‘(C D) X :TEST EQUAL) = ((C D) E)

- (SETQ X ‘((B) C))
- (SETQ Y (CONS ‘A X))
- (MEMBER (CAR X) Y) = ((B) C)
- (MEMBER ‘(B) Y) = NIL
- (MEMBER (CADR Y) X) = ((B) C)

4. Predicates for numbers

NUMBERP subr 1 (e): T, NIL

Check if it's a number or not.

ZEROP subr 1 (n): T, NIL

Check if n is the number 0 or not. If the argument is not evaluated by number the result is undefined or error.

PLUSP subr 1 (n): T, NIL

Check if n is a strictly positive number.

MINUSP subr 1 (n): T, NIL

Check if n is a strictly negative number.

5. Arithmetic operations

+ fsubr 1, ... (... n ...): n

The arguments are evaluated at the numerical values n1, ..., nk. The returned result is n1 + ... + nk or error if one of the arguments is not evaluated at a numeric value.

- fsubr 1, ... (... n ...): n

The arguments are evaluated at the numerical values n1, ..., nk. The returned result is n1 - ... - nk or error if one of the arguments is not evaluated at a numeric value.

*** fsubr 1, ... (... n ...): n**

The arguments are evaluated at the numerical values n1, ..., nk. The returned result is n1 * ... * nk or error if one of the arguments is not evaluated at a numeric value.

/ fsubr 1, ... (... n ...): n

The arguments are evaluated at the numerical values n1, ..., nk. The returned result is n1 / ... / nk or error if one of the arguments is not evaluated at a numeric value.

1+ subr 1 (n): n
1- subr 1 (n): n

The argument is evaluated at the numeric value n. The returned result is n + 1 (respectively n - 1), or error if the argument is not evaluated at a numeric value.

MAX fsubr 1, ... (... n ...): n

MIN fsubr 1, ... (... n ...): n

The returned result is the maximum (respectively the minimum) of the numeric values at which the arguments are evaluated, or error if the argument is not evaluated at a numeric value.

6. Logical operations

NOT subr 1 (e): T, NIL

Returns T if argument e is evaluated at NIL; otherwise, return NIL. It's equivalent to the NULL function.

AND fsubr 0, ... (... e ...): e

It is evaluated from left to right until the first NIL, in which case NIL is returned; otherwise the result is the value of the last argument.

OR fsubr 0, ... (... e ...): e

It is evaluated from left to right to the first element evaluated at a value other than NIL, in which case that value is returned; otherwise the result is NIL.

Remark: AND and OR do not evaluate all parameters unconditionally. They only evaluate from left to right until a decision on the result can be made. At that point the functions end and the rest of parameters are not evaluated. AND and OR are thus special operators.

7. Relational operators for numbers

= subr 2 (n n): T, NIL

< subr 2 (n n): T, NIL

<= subr 2 (n n): T, NIL

> subr 2 (n n): T, NIL

>= subr 2 (n n): T, NIL

They have the usual meaning. The parameters evaluate to numerical values.

8. Defining user functions. DEFUN function

DEFUN fsubr 3, ... (s l f ...): s

The DEFUN function creates a new function named as the first argument (symbol s), and as formal parameters the symbol elements of the list that constitutes the second argument; the body of the created function consists of one or more forms, as arguments, on the third and possibly the following positions (the evaluation of these forms at execution represents the side effect). Returns the name of the function created. The DEFUN function does not evaluate any arguments.

The call of a function defined by

```
(DEFUN fname (p1 p2 ... pn)
        ...
)
```

it is a form

```
(fname arg1 arg2 ... argn)
```

where **fname** is a symbol, and **argi** are forms; the evaluation of the call proceeds as follows:

(a) the arguments arg1, arg2, ... argn are evaluated; let v1, v2, ... vn be their values;
(b) each formal parameter in the function definition is bound to the value of the corresponding argument in the call (p1 to v1, p2 to v2, ..., pn to vn); if at the time of the call the symbols representing formal parameters already had values, they are saved for later restoration;
(c) each form in the body of the function is evaluated in order, the value of the last form being returned as the value of the function call;

(d) the former values of the formal parameters are restored, ie p1, p2 ... pn is "unbound" from the values v1, v2, ... and is bound again to the corresponding saved values (if applicable).

- (DEFUN SECOND (X) (CAR (CDR X))) defines a function that selects the second item in the X list;
- (DEFUN AD1 () (+ 1 X)) defines a function without parameters that returns a value 1 greater than the value at which the global variable X is evaluated.

9. Branching of processing. COND function

The COND function is similar to the CASE or SWITCH selectors in Pascal and C, respectively.

COND fsubr 0, ... (... l ...): e

In the description above it is a list nonempty of arbitrary length (f1 f2 ... fn) called clause. COND admits as many clauses as arguments and as many forms in a clause. Here is how the COND function works:

- the clauses are examined in turn in the order of their appearance in the appeal, evaluating only the first element of each clause until a different one from the NIL is encountered. The respective clause will be selected and the order f2, f3, ... fn will be evaluated in order. Returns the value of the last evaluated form from the selected clause;
- if no clause is selected, COND returns NIL;
- if a clause consisting of a single element has been selected, the value of that element is returned (so the test and result elements can be the same; so in the clauses of the form (something T) the element T may be missing).

Consider the next sequence

```
• (SETQ X 10)                                   X is evaluated at 10
• (SETQ Y
     (COND
          ((> X 5) (SETQ Z X) (CONS ʻA ʻ(B)))   Z is evaluated at 10
          (T ʻA)
     )
  )                                             Y is evaluated at (A B)
```

The following example returns the argument if it is an atom, NIL if it is an empty list and the first element if the argument is a list.

```
(DEFUN FIRST (X)
        (COND
                ((ATOM X) X)
                ((NULL X) NIL); useless
                (T (CAR X))
        )
)
```

The following example returns the maximum values of the two arguments.

```
(DEATH MAX (X Y)
        (COND
                ((> X Y) X)
                (T Y)
        )
)
```

The following example returns the last item in a list, superficially.

```
(DEFUN LAST (X)
        (COND
                ((ATOM X) X)
                ((NULL (CDR X)) (CAR X))
                (T (LAST (CDR X)))
        )
)
```

The following example rewrites CAR to return NIL if the argument is atom and does not produce an error message.

```
(DEFUN XCAR (X)
        (COND
                ((ATOM X) NIL)
                (T (CAR X))
        )
)
```

Remark. DEFUN may also redefine standard system functions. For example, if the CAR function is redefined as follows: (DEFUN CAR (L) (CDR L)), then (CAR ′(1 2 3)) will be evaluated at (2 3).

10. Examples of definitions of system functions

The following example shows a possible definition for the APPEND function.

```
(DEFUN APPEND (L1 L2)
        (COND
                ((NULL L1) L2)
                (T (CONS (CAR L1)) (APPEND (CDR L1) L2))); copy the L1 list
        )
)
```

The following example shows a possible definition for the MEMBER function.

```
(DEFUN MEMBER (ELEM LIST)
        (COND
                ((ATOM LIST) NIL)
                ((EQUAL ELEM (CAR LIST)) LIST)
                (T (MEMBER ELEM (CDR LIST)))
        )
)
```

11. Other examples

EXAMPLE 1 Calculate the sum of numerical atoms at any level in a nonlinear list.

Recursive models

Version 1

$$suma(l_1 l_2 \ldots l_n) = \begin{cases} 0 & daca\ lista\ e\ vida \\ l_1 + suma(l_2 \ldots l_n) & daca\ l_1\ este\ atom\ numeric \\ suma(l_2 \ldots l_n) & daca\ l_1\ este\ atom \\ suma(l_1) + suma(l_2 \ldots l_n) & altfel \end{cases}$$

Version 2

$$suma(l) = \begin{cases} l & dacă\ l\ atom\ numeric \\ 0 & dacă\ l\ atom \\ suma(l_1) \oplus suma(l_2 \ldots l_n) & altfel, l = (l_1 l_2 \ldots l_n)\ e\ lista \end{cases}$$

(sum ‘(1 (2 a (3 4) b 5) c 1)) → 16

EXAMPLE 2 Build the list obtained by adding an item at the end of a list.

```
(add ‘3 ‘(1 2)) → (1 2 3)
(add ‘(3) ‘(1 2)) → (1 2 (3))
(add ‘3 ‘()) → (3)
```

Recursive model

; return list ($l_1, l_2, \ldots, l_n, e$)

$$adaug(e, l_1 l_2 \ldots l_n) = \begin{cases} (e) & daca \quad l \ e \ vida \\ l_1 \oplus adaug(e, l_2 \ldots l_n) & altfel \end{cases}$$

```
(defun add(e l)
        (cond
                ((null l) (list e)) ; (list e) or (cons e nil)
                 (t (cons (car l) (add e (cdr l))))
        )
)
```

EXAMPLE 4.3. **The method of the collector variable.** Define a function that reverses a linear list.

(invers ‘(1 2 3)) will return (3 2 1).

Recursive model

$$invers(l_1l_2\ldots l_n) = \begin{cases} \phi & \textit{if } l \textit{ is empty} \\ invers(l_2\ldots l_n) \otimes l_1 & \textit{otherwise} \end{cases}$$

A possible definition for the REVERSE function is:

```
      (DEFUN INVERS (L)
             (COND
                   ((ATOM L) L)
                  (T (APPEND (INVERS (CDR L)) (LIST (CAR L))))
             )
      )
```

The time complexity is given by the recurrence

$$T(n) = \begin{cases} 1 & daca\ n = 0 \\ T(n-1) + n & altfel \end{cases}$$

The problem is that such a definition consumes a lot of memory. The efficiency of lisp function application (expressed by memory consumption) is measured by the number of CONS calls it performs. Let us recall that (LIST arg) is equivalent to (CONS arg NIL) and let us also point out that APPEND works by copying the first argument, which is then "glued" to the second argument. Thus, the INVERS function defined above will perform the copying of each one (INVERS (CDR L)) before "pasting" it to the second argument. For example for the list (A B C D E) will copy the lists NIL, (E), (E D), (E D

C), (E D C B), so for a list of size N we will have 1 + 2 + ... + (N-1) = N (N-1) / 2 uses of CONS. So the complexity of time is $\theta(\boldsymbol{n^2})$, n being the number of items in the list.

One solution to reduce the complexity of the time of the reversal operation is **to use the method of the collector variable:** writing an auxiliary function that uses two parameters (Col = destination list and L = source list), its purpose being to pass one element in turn from L to the Col:

L	Col
(1, 2, 3)	Ø
(2, 3)	(1)
(3)	(2, 1)
Ø	(3, 2, 1)

Recursive models

$$invers\_aux(l_1 l_2 \dots l_n, Col) = \begin{cases} Col & daca\ l\ e\ vida \\ invers\_aux(l_2 \dots l_n, l_1 \oplus Col) & altfel \end{cases}$$

$$invers(l_1 l_2 \dots l_n) = invers\_aux(l_1 l_2 \dots l_n, \emptyset)$$

```
(DEFUN INVERS_AUX (L Col)
        (COND
                ((NULL L) Col)
               (T (INVERS_AUX (CDR L) (CONS (CAR L) Col)))
         )
 )
```

What we want is achieved by the call (INVERS_AUX L ()), but let's not forget that we started from the need to define a function to be called with (INVERS L). Therefore, the INVERS function will be defined:

```
(DEFUN INVERS (L)
        (INVERS_AUX  L ())
 )
```

The INVERS_AUX function has the role of auxiliary function, emphasizing the role of the **Col** argument as a **collector variable** (variable that collects the partial results until the final result is obtained). As many CONS will be performed as the length of the source list L, so the complexity of time is θ***(n)***, *n* being the number of items in the list.

$$T(n) = \begin{cases} 1 & daca\ n = 0 \\ T(n-1) + 1 & altfel \end{cases}$$

Let us note, therefore, that collecting the partial results in a separate variable can contribute to the reduction of the complexity of the algorithm. But this is not true in all cases: there are situations where the use of a collector variable increases the complexity of processing, if the elements are added at the end of the collector (not at its beginning, as in the example indicated).

EXAMPLE 4. Define a function that determines the list of pairs between a given element and the elements of a list.

(LISTA ‘A ‘(B C D)) = ((A B) (A C) (A D))

<u>Recursive model</u>

$$lista(e, l_1 l_2 \dots l_n) = \begin{cases} \emptyset & daca\ l = \emptyset \\ (e, l_1) \oplus lista(e, l_2 \dots l_n) & altfel \end{cases}$$

```
(DEFUN LISTA (E L)
        (COND
                ((NULL L) NIL)
                (T (CONS (LIST E (CAR L)) (LISTA E (CDR L))))
        )
)
```

EXAMPLE 5. Define a function that determines the list of pairs of items in strictly ascending order that can be formed with the elements of a numerical list (keep the order of the items in the list).

(perechi '(3 1 5 0 4)) = ((3 5) (3 4) (1 5) (1 4))

We will use an auxiliary function that returns the list of pairs of items in strictly ascending order, which can be formed between an item and the elements of a list.

(per '2 '(3 1 5 0 4)) = ((2 3) (2 5) (2 4))

$$per(e, l_1 l_2 \dots l_n) = \begin{cases} \emptyset & daca\ l = \emptyset \\ (e, l_1) \oplus per(e, l_2 \dots l_n) & dacă\ e < l_1 \\ per(e, l_2 \dots l_n) & altfel \end{cases}$$

```
(defun per (e l)
    (cond
        ((null l) nil)
        (T (cond
            ((< e (car l)) (cons (list e (car l))(per e (cdr l))))
            (T (per e (cdr l)))
           )
        )
    )
)
```

$$perechi(l_1 l_2 \dots l_n) = \begin{cases} \emptyset & daca\ l = \emptyset \\ per(l_1, l_2 \dots l_n) \oplus perechi(\ l_2 \dots l_n) & altfel \end{cases}$$

```
(defun perechi (l)
  (cond
     ((null l) nil)
     (t (append (per (car l) (cdr l)) (perechi (cdr l))))
   ) )
```

EXAMPLE 6. A nonlinear list is given. It is required to double the numerical values at any level of the list, keeping its hierarchical structure.

(dublare '(1 b 2 (c (3 h 4)) (d 6)))) → (2 b 8 (c (6 h 8)) (d 12)))

Version 1

$$dublare(l_1 l_2 \dots l_n) = \begin{cases} \emptyset & daca\ l = \emptyset \\ 2l_1 \oplus dublare(l_2 \dots l_n) & dacă\ l_1\ numeric \\ l_1 \oplus dublare(l_2 \dots l_n) & l_1\ atom \\ dublare(l_1) \oplus dublare(l_2 \dots l_n) & altfel \end{cases}$$

```
(defun dublare(l)
        (cond
                ((null l) nil)
              ((numberp (car l)) (cons (* 2 (car l)) (dublare (cdr l))))
              ((atom (car l)) (cons (car l) (dublare (cdr l))))
              (t (cons (dublare (car l)) (dublare (cdr l))))
         )
)
```

Version 2

$$dublare(l) = \begin{cases} 2l & dacă\ l\ numar \\ l & dacă\ l\ atom \\ dublare(l_1) \oplus dublare(l_2 \dots l_n) & altfel, l = (l_1 l_2 \dots l_n) e\ lista \end{cases}$$

```
(defun dublare(l)
      (cond
            ((numberp l) (* 2 l))
            ((atom l) l)
            (t (cons (dublare (car l)) (dublare (cdr l))))
       )
)
```

EXAMPLE 7. ***Using a collector variable can increase complexity.*** A linear list is given. What is the effect of the following assessment:

(lista ′(1 a 2 b 3 c)) → ???

Write the mathematical models for each function.

Version 1 – directly recursive (no collector variable)

```
(defun lista (l)
  (cond
    ((null l) nil)
    ((numberp (car l)) (cons (car l) (lista (cdr l))))
    (t (lista (cdr l)))
  )
)
```

What is the worst case time complexity?

Version 2 – with collector variable

```
(defun lista_aux (l col)
   (cond
     ((null l) col)
     ((numberp (car l)) (lista_aux (cdr l) (append col (list (car l)))))
     (t (lista_aux (cdr l) col))
   )
)

(defun lista (l)
   (lista_aux l nil)
)
```

What is the worst case time complexity?

EXAMPLE 8. A linear list is given. What is the effect of the function TRAVERSE?

```
(defun traverse_aux(L k col)
        (cond
            ((null L) nil)
            ((= k 0) (list col L))
            (t (traverse_aux (cdr L) (- k 1) (cons (car l) col)))
        )
)

(defun traverse (L k)
      (traverse_aux L nil)
)

(traverse '(1 2 3 4 5) 3) → ((3 2 1) (4 5))
```