LECTURE 12. Iterative forms. Optional arguments. Macrodefinitions. Data structures

Contents

Despite its elegance as an algorithmic description method, recursion is not always the most appropriate method of solving problems. This is because the resources consumed by the implementation of recursion are usually very large. Therefore, the Lisp language provides a set of iterative forms that allow the control of execution in a style close to that known in classical imperative languages. These forms are: PROG (equivalent of begin-end instruction blocks, with the possibility of defining local variables in the block), WHILE, UNTIL and DO (equivalents of WHILE, REPEAT and FOR instructions respectively).

1. PROG and its variants

PROG fsubr 1, ... (l f1 ... fn): e

Here l is an initialization list containing elements of the form "(variable value)" or only "variable". In the first case the variable is initialized with the specified value, and in the second case it gets the NIL value. The variables are temporary and local to PROG. The body of the PROG function consists of the forms f1, ..., fn which are evaluated sequentially. At the end of the PROG function body, NIL is returned by default. The following form must be used to return another value

RETURN subr 1 (e): e

The forms can also be represented by atoms (symbolic or numerical) in which case they are interpreted as labels, and are not evaluated. Labels are used to resume the evaluation of PROG from a specified point. This transfer of control is done by

GO nsubr 1 (f)

where f is a label atom or a form evaluable to a label atom.

A variant of defining (also recursively) the generator function from an earlier lecture with the help of the PROG form is:

```
(DEFUN GENCAR1 (F L)
    (PROG (R)
        (COND
            ((NULL L) NIL)
            ((SETQ R (FUNCALL F (CAR L)))
                (RETURN (CONS R (GENCAR1 F (CDR L)))))
        )
    )
)
```

This definition can be reformulated using explicit cycling and thus avoiding recursion:

```
(DEFUN GENCAR1 (F L)
    (PROG (R REZ) ; both variables are LOCAL; initialized with NIL
        CYCLE
        (COND
            ((NULL L) NIL)
            (T    (SETQ R (FUNCALL F (CAR L)))
                  (SETQ REZ (CONS R REZ) L (CDR L))
                  (GO CYCLE))
        )
        (RETURN (REVERSE REZ))
    )
)
```

Remark. The label does not have to be local to the PROG that has the form (GO label), but then it must be found in an external PROG that contains the original one. So nested PROGs are allowed, but let's not forget that a label can only appear inside a PROG form.

As an example, here is a way to build a function to calculate the length of a list:

```
(DEFUN LENGTH (L)
        (PROG ((LUNG 0))
                CYCLE
                (COND
                        ((NULL L) (RETURN LUNG))
                        (T      (SETQ LUNG (+ LUNG 1))
                                (SETQ L (CDR L))
                                (GO CYCLE)
                        )
                )
        )
)
```

PROGi fsubr 0- (f1 ... fn): e

As variants of the PROG form, the LISP language provides the PROG1, PROG2, and PROGN forms with no local variables, that perform the evaluation of the specified sequence of forms by returning the value of the first, second, or last evaluated form (as opposed to the default NIL value of the PROG form). Am example for these forms follows.

```
(SETQ X (PROG1 Y (SETQ Y X)))
```

switches the values of symbols X and Y (the list of initializations no longer appears in these forms). Note also that the value of Y evaluated at the input to PROG1 is returned and not the value of Y affected by the side effect of the assignment (SETQ Y X).

2. DO and its variants

A second possibility to implement iterativity is by using the DO form and its variants. DO form has syntax:

DO fsubr 2, ... (l l f1 ... fn): e

The call of the DO function has the form

(DO list_init list_clause forms)

where the three arguments have the following meaning:

(i) the index specification (list_init) that has the form

```
(       (var1 val_init1 step1)
        ...
        (varn val_initn stepn)
)
```

the effect being to bind the various symbols to the corresponding val_initi initial values, or NIL if these are not specified; note that this binding operation takes place at the same time for all variables;

(ii) the cycle completion specification (test clause, clause_list) that has the form

(test_terminate g1 ... gn)

(iii) the body of the cycle, made up of forms that will be evaluated repeatedly (f1 ... fn) as an effect of cycling

An iterative cycle is as follows:

(1) The form test_termination is evaluated (so it is a cycle with initial test, of type WHILE). At the evaluation other than NIL of the clause test, the output forms gi will be evaluated in order and the result of the last evaluation, gn, will be returned, as in the case of the PROGN form. Note that these forms will be evaluated only once, ie at the end of the cycle. If there is no gi form then NIL will be returned. Note that the termination specification has the same syntax as a COND clause.
(2) If test_termination is evaluated at NIL then we proceed to evaluate the forms that form the body of DO, ie f1, ..., fn.
(3) When the end of the body of the function is met, each variable index is assigned the value of the corresponding step update forms. If the step form is missing, the vari variable will no longer be updated, remaining with the value it has. Note also here that the assignment operation takes place at the same time for all variables. The cycle is repeated starting with (1).

The DO control structure is considered to be executed within a PROG block, which makes it possible to exit the DO at any time by evaluating a RETURN form.

The following example shows a variant of implementing the REVERSE function:

```
(DEFUN REVERSE (LIST)
        (DO    ((L LIST (CDR L))     ; initialization of cycling symbols
                (RESULT NIL)         ; functions as a collector variable
                )
                ((NULL L) RESULT)            ; test clause
                (SETF RESULT (CONS (CAR L) RESULT)); body of forms
        )
)
```

The following example calculates the length of a list:

```
(DEFUN LONG (LIST)
        (DO    ((L LIST (CDR L))     ; initialization of cycling variables
                (RESULT 0 (+ RESULT 1)); functions as a collector variable
                )
                ((NULL L) RESULT); test clause
        )      ; note that this function has no body of forms
)
```

A variant of the DO function is the DO* function. Unlike DO, which evaluates cycling variables at the same time, DO* evaluates cycling variables sequentially, in the order they are specified.

DOTIMES fsubr 2- (l f1 ... fn): e

The DOTIMES function is a variant of the DO function equivalent to FOR in imperative languages. It is called in the form

```
(DOTIMES (variable counter result) body)
```

Here's how a DOTIMES call works:

(1) the variable given as the first parameter of the first list is initialized to zero;
(2) if the value of the variable is equal to the value of the counter, then the resulting form is evaluated; the value obtained is the result of the DOTIMES function;
(3) the forms that represent the body of the function are evaluated;
(4) the value of the variable is increased by one;
(5) returns to point (2).

As we may see, this DOTIMES is equivalent to the following Pseudocode sequence:

```
FOR variable = 0 TO counter-1 EXECUTE
        body
RESULT result
```

DOLIST fsubr 2- (l f1 ... fn): e

The DOLIST function is similar to the DOTIMES function. It is called like

(DOLIST (variable list result) body)

Unlike DOTIMES, where the variable receives numeric values between 0 and the counter value, in the case of the DOLIST function the variable receives in turn the values of all the elements in the list given on the second position of the first list.

3. Functions with optional arguments

In the list of formal parameters of a function we can use the following variables:

- &OPTIONAL - if it appears in the list of formal parameters then the next formal parameter will take as value the corresponding parameter from the list of current parameters, or NIL if the current parameter does not exist;
- &REST - if it appears in the list of formal parameters then the next formal parameter will take as value the list of current parameters left unassigned, or NIL if there are no more current unassigned parameters.

Examples

1.
```
(DEFUN EX1 (X &OPTIONAL Y)
    (COND
        ((NULL Y) X)
        (T (+ X Y))
    )
)
```

- (EX1 1 2) is evaluated at 3
- (EX1 1) is evaluated at 1

2.
```
(DEFUN EX2 (X &REST Y)
    (+ X (APPLY #'+ Y))
)
```

- (EX2 1 2 3) is evaluated at 6
- (EX2 4 5) is evaluated at 9
- (EX2 3) is evaluated at 3

3. (DEFUN EX3 (L1 &REST L2)

(COND

((NULL (CAR L2)) NIL)

(T (CONS (MAPCAR #'* L1 (CAR L2))

(APPLY #'EX3 (CONS L1 (CDR L2)))

))

)

)

- (EX3 '(1 2) '(3 4) '(5 6)) is evaluated at ((3 8) (5 12))

Suppose we want to construct a function that adds 1 to the value of an expression:

(DEFUN INC (NUMBER)

(+ NUMBER 1)

)

This function will be used as follows:

- (INC 10) will be evaluated at 11

Of course, as much as we want, we can add similar functions for other increment values. Alternatively, we can rewrite the INC function to accept a second parameter:

(DEFUN INC (NUMBER INCREMENT)

(+ INCREMENT NUMBER)

)

In the vast majority of cases, however, we will need to increment by 1 the value of the expression corresponding to NUMBER. In this situation we can use the facility of optional arguments. Here is the definition of the INC function with an optional argument:

```
(DEFUN INC (NUMBER &OPTIONAL INCREMENT)
        (COND
                ((NULL INCREMENT) (+ NUMBER 1))
                (T (+ INCREMENT NUMBER))
        )
)
```

The &OPTIONAL character indicates that &OPTIONAL is a parameter separator, not a parameter itself. The parameters following &OPTIONAL are related to the entry in the procedure in the same way as the other parameters. If there is no corresponding current parameter, the value of an optional formal parameter is considered NIL. Here are some examples of using the INC function we just built:

- (INC 10) is evaluated at 11
- (INC 10 3) is evaluated at 13
- (INC (* 10 2) 5) is evaluated at 25

We can also consider a default value for the optional parameters. Here is a definition of the INC function in which the optional parameter has the default value 1:

```
(DEFUN INC (NUMBER &OPTIONAL (INCREMENT 1))
        (+ INCREMENT NUMBER)
)
```

Note that this time instead of the optional parameter, a list of two elements is specified: the first element is the name of the optional parameter, and the second element is the value with which it is initialized when the corresponding current argument is not present.

Note that we can have as many optional arguments. All these arguments will be passed in the parameter list after the &OPTIONAL symbol, which will appear only once. We can also have a single optional argument signaled by &REST, whose value becomes the list of all given arguments except those that correspond to the required and optional parameters. Let a new version of the INC function:

```
(DEFUN INC (NUMBER &REST INCREMENT)
        (COND
                ((NULL INCREMENT) (+ NUMBER 1))
                (T (+ NUMBER (APPLY #'+ INCREMENT)))
        )
)
```

Here are some examples of how to apply this feature:

- (INC 10) is evaluated at 11
- (INC 10 1 2 3) is evaluated at 16

Remark. So the complete way to define a function is as follows:

(DEFUN name ([par_1 ... par_n] [&OPTIONAL par_{n+1} [... par_m]] [&REST par_{m+1}])
    body
)

If n = 0, the function thus defined has no mandatory parameters, it can be called with zero or more parameters. If the function thus defined has optional parameters, if fewer current parameters are specified on the call than the corresponding formal parameters, the values of the current parameters are assigned to the formal parameters from left to right. If the function thus defined has an optional parameter, it will have as value the list of all current parameters remaining after solving the mandatory and optional parameters.

As an example, the following function calculates the sum of numbers given as parameters:

```
(DEFUN SUM (E &REST L)
        (COND
                ((NULL L) E)
                (T (+ E (APPLY #'SUM L)))
        )
)
```

Remark. This function can be called with at least one parameter. Modify the function so that it can be called without parameters.

Remark. The value of the first element given as a parameter will be assigned to the formal parameter E, and the list of values of all other parameters will be assigned to the formal parameter L. In order to apply the SUM function to the elements of the L list, we will need the APPLY function. The result of this call must be added to the value of E.

4. Macro-definitions

From a syntactic point of view, the macro-definitions are constructed in the same way as the functions, with the difference that instead of using the DEFUN function, the DEFMACRO function will be used:

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

The DEFMACRO function creates a macro definition with the name of the first argument (symbol s), and as formal parameters the symbolic elements of the list that constitutes the second argument; the body of the created macro-definition consists of one or more forms located, as arguments, on the third position and possibly on the following ones. The value returned as a result is the name of the created macro. The DEFMACRO function does not evaluate any arguments.

The way the macro-definitions created with DEFMACRO work is different from that of the functions created with DEFUN:

- macro-definition parameters are not evaluated;
- the body of the macro-definition is evaluated and an intermediate S-expression will occur;
- this S-expression will be evaluated again, this being the moment when the parameters are evaluated.

Let's look at a comparative example first. Be the followingdefinitions:

```
(DEFMACRO DEMO-MACRO (PAR)
        (PRINT PAR)
)
```

```
(DEFUN DEMO-FUN (PAR)
        (PRINT PAR)
)
```

Consider this value binding

```
(SETF PARAM 'VALUE)
```

The effect of the evaluation (DEMO-MACRO PARAM) will be

```
PARAM
VALUE
```

because at first no attempt is made to evaluate the PARAM parameter. As such, PRINT will print PARAM, after which it will return PARAM as the value of the intermediate form of the macro-definition. Then this intermediate value is evaluated and VALUE will occur.

If we now evaluate the form (DEMO-FUN PARAM) the effect will be

```
VALUE
VALUE
```

because the PARAM parameter is evaluated from the beginning, PRINT will print its value, and return it as the value of the function call.

As another example, we want to produce an INC macro-definition that increases by 1 the value of its parameter, as in the following example:

- (SETF X 10) is evaluated at 10
- (INC X) will produce 11
- X is evaluated at 11

This means that the intermediate form will have to be

(SETF parameter (+ 1 parameter))

Here are two possibilities:

```
(DEFMACRO INC (N)
        (LIST 'SETF N (LIST '+ 1 N))
)

(DEFMACRO INC (N)
        (SUBST N 'N '(SETF N (+ 1 N)))
)
```

5. Reverse apostrophe (backquote)

The reverse apostrophe (`) makes writing macros much easier. There is a way to create expressions in which most of it is fixed, and in which only a few variable details need to be filled in. The effect of the inverse apostrophe is similar to that of the apostrophe ('), in that it blocks the evaluation of the following S-expression, except that any comma (,) that appears will produce the evaluation of the following expression. Here is an example:

- (SETF VARIABLE EXAMPLE)
- `(THIS IS ,VARIABLE) is evaluated at (THIS IS EXAMPLE)

Also, the inverse apostrophe accepts the comma operator and accompanied by an @ sign. This combination also produces the evaluation of the following expression, but with the difference that the resulting value must be a list. Items in this list are dissolved in the list in which the combination appears, @. Here is an example:

- (SETF VARIABLE '(OTHER EXAMPLE))
- `(THIS IS ONE ,VARIABLE) is evaluated at
  (THIS IS ONE (OTHER EXAMPLE))
- `(THIS IS ONE ,@VARIABLE) is evaluated at
  (THIS IS ONE OTHER EXAMPLE)

Using the inverse apostrophe, the INC macro-definition will be written simpler as follows:

```
(DEFMACRO INC (N)
        `(SETF ,N (+ 1 ,N))
)
```

Note also that the use of the inverse apostrophe facilitates the printing of intermediate results.

6. Property lists

A symbolic atom may have attached a lot of properties, as well as the values of these properties. Syntactically, properties are symbolic atoms. There are five functions that allow access to the list of properties of a symbol: PUTPROP, GET, SYMBOL-PLIST, REMPROP and SETPLIST.

PUTPROP subr 3 (symbolic-atom value property): s

The PUTPROP function evaluates all its arguments, attaching to the symbolic atom the property with the specified name and value. If the property for the symbolic atom already exists, then its value will be updated with the specified one. The result returned by PUTPROP is the symbolic atom (evaluated). We mention that, in general, properties are symbols, although, syntactically, they can be S-expressions.

GET subr 2 (symbolic-atom property): e

The GET function evaluates all its arguments, returning the value of the property attached to the symbolic atom or NIL, if this property does not exist. Note that the GET function in GCLisp uses the EQL predicate to test equality (in general, properties are symbols and as a result this is irrelevant).

SYMBOL-PLIST subr 1 (symbolic-atom): l

The SYMBOL-PLIST function evaluates its argument, returning a list of properties of the symbolic atom, in the form (prop1 val1 prop2 val2 ..... propn valn).

REM-PROP subr 2 (symbolic-atom property): T, NIL

The REM-PROP function evaluates all its arguments, having the effect of deleting the specified property attached to the symbolic atom. The function returns NIL if the symbolic atom does not have the specified property, or T otherwise.

SETPLIST subr 2 (symbolic-atom list): s

The SETPLIST function evaluates all its arguments, and the list must be evaluated to a list of form (prop1 val1 prop2 val2 ..... propn valn). The function attaches to the symbolic atom the properties prop1, prop2, ..., propn, with the values val1, val2, ..., valn. The result returned by SETPLIST is the symbolic atom (evaluated).

Remark:

CLISP does not have PUTPROP. This can, however, be simulated that by creating a macrodefinition:

```
(defmacro putprop (var val prop)
  `(setf (get ,var ,prop) ,val)
)
```

Examples:

(1) Consider the following definitions of properties:

```
(PUTPROP 'Ion ‘Mihai ‘father)          ; father (Ion) = Mihai
(PUTPROP 'Ion ‘Maria ‘mother)          ; mother (Ion) = Maria
(PUTPROP 'Mihai ‘Gheorghe ‘father)     ; father (Mihai) = Gheorghe
(PUTPROP 'Mihai ‘Ana ‘mother)          ; mother (Mihai) = Ana
```

```
(PUTPROP 'Maria ‘George ‘father)                ; father (Maria) = George
(PUTPROP 'Maria ‘Mihaela ‘mother)               ; mother (Maria) = Mihaela
```

If we want to get the list of Ion's grandparents we will do the following::

```
(MAPCAR #'GET (LIST     (GET 'Ion ‘father) (GET 'Ion ‘mother)
                        (GET 'Ion ‘father) (GET 'Ion ‘mother)     )
        '(father father mother mother)
)       ; returns (Gheorghe George Ana Mihaela)
```

If we wish that the symbolic atom **Dan** to have all the properties of the symbolic atom **Ion** we will evaluate the following function:

```
(SETPLIST 'Dan (SYMBOL-PLIST 'Ion))
```

The effect of this assessment will be that **Dan** he has the same parents (mother and father properties) as **Ion**.

```
(GET 'Dan ‘father)          ; returns MIHAI
```

(2) Consider the following definitions of properties:

```
(PUTPROP 'Ion ‘Mihai 'parent)               ; parent (Ion) = Mihai
(PUTPROP 'Mihai ‘George 'parent)            ; parent (Michael) = George
(PUTPROP 'George ‘Emil 'parent)             ; parent (George) = Emil
(PUTPROP 'Mihaela ‘Dan 'parent)             ; parent (MIhaela) = Dan
(PUTPROP 'Dan ‘George 'parent)              ; parent (Dan) = George
```

The ANCESTOR function whose definition we give below provides as a result the ancestors of the symbolic atom a in relation to the property prop.

```
(DEFUN ANCESTOR (a prop)
        ((LAMBDA (v)
                (COND
                        ((NULL v) NIL)
                        (T (CONS v (ANCESTOR v prop)))
         )
         ) (GET a prop))
)
```

As a result of this definitions, the call (ANCESTOR 'Ion 'parent) provides the list of Ion's ancestors, ie (MIHAI GEORGE EMIL) - the list of ancestors is generated in ascending order, ie the direct parent, then the parent's father, and so on.

Considering the same definitions of properties as above, we aim to define a function (COMMON a b prop) that determines the closest common ancestor (from the point of view of a property defined prop) of two symbolic atoms a and b.

We will use a helper function (PRIM I1 I2) that determines the first common element of lists I1 and I2, respectively NIL if the two lists do not have common elements.

```
(DEFUN PRIM (I1 I2)
        (COND
                ((NULL I1) NIL)
                ((MEMBER (CAR I1) I2) (CAR I1))
                (T (PRIM (CDR I1) I2))
        )
)
```

Using the PRIM function, the COMMON function will be described below:

```
(DEFUN COMMON (a b prop)
        (FUNCALL #'PRIM (ANCESTOR a prop) (ANCESTOR b prop))
)
```

According to this definition, we can find the nearest common ancestor according to the parent property defined above.

```
(COMMON 'Mihaela ‘Ion 'parent)          ; returns GEORGE
(COMMON 'Ion ‘Emil 'parent)             ; returns NIL
```

7. Association lists

The association list is a list of pairs (NAME . VALUE), where VALUE can be any form.

ASSOC subr 2 (key assoc-list): pp

The ASSOC function evaluates all its arguments and looks in the parameter association list for the pair with the corresponding key as the left part of the CONS cell. The search is performed with EQL, and the found pair will be returned as a result. If no pair is found that has the form value as a secondary field, NIL is returned.

```
(SETF L (LIST (CONS 'A 'B) (CONS 'C 'D)))      ; L will be ((A . B) (C . D))
(ASSOC 'C L)                                   ; returns (C . D)
```

ASSOC may be used with SETF in updating values in association lists.

```
(SETF L (LIST (CONS 'A ' B) (CONS 'C 'D)))     ; L will be ((A . B) (C . D))
(SETF (CDR (ASSOC 'C L)) 'X)                   ; replaces (C . D) with (C . X)
```

RASSOC subr 2 (expr assoc-list): pp

The RASSOC function evaluates all its arguments and looks in the parameter association list for the pair that has the value of the form as the right part of the CONS cell. The search is performed with EQL, and the found pair will be returned as a result. If no pair is found that has the form value as a secondary field, NIL is returned.

```
(SETF L (LIST (CONS 'A 'B) (CONS 'C 'D)))      ; L will be ((A . B) (C . D))
(RASSOC 'D L)                                  ; returns (C . D)
```

Other relevant functions

(SETF L (**PAIRLIS** ' (A B C) ' (1 2 3))) ; L will be ((A . 1) (B . 2) (C . 3))

(SETF L (LIST (CONS 'A '1) (CONS 'C '2))) ; L will be ((A . 1) (C . 2))

(SETF L (**ACONS** 'X '0 L)) , L will be ((X . 0) (A . 1) (C . 2))

8. Records

A record type structure is defined using the following function:

DEFSTRUCT fsubr 3, ... (sl ...): s

The DEFSTRUCT function defines a structure (record) with the name of the first argument (symbol s), the following arguments being fields of the record, field, or pairs of form (field initial_value). Returns the name of the created structure. The DEFSTRUCT function does not evaluate any arguments.

An example of defining a record in which fields are initialized with values:

(DEFSTRUCT structure_name (field_name$_1$ initial_value$_1$)
(field_name$_2$ initial_value$_2$)
...
(field_name$_n$ initial_value$_n$)
)

An example of defining a record where the fields have no initial values:

(DEFSTRUCT structure_name field_name$_1$
Field_name$_2$
...
field_name$_n$
)

If the fields did not specify initial values, they are 0 by default.

The DEFSTRUCT function automatically generates a function to create instances of the structure named by this definition. For the previously defined

structure, the function is called **make-structure_name**, and in general the instance builder name is **make-*defstructname***.

As a result, the construction of a variable with the structure specified by the previous DEFSTRUCT definition is as follows:

(SETF variable (MAKE-*structure_name*))

Returning the field value of a variable of a structure defined with DEFSTRUCT is done as follows:

(*structure-name_field-name* variable)

Assigning a field value of a variable of a structure defined with DEFSTRUCT is done as follows:

(SETF (*structure-name_field-name* variable) value)

Example:

Consider the definition of a type structure **student** with the following fields: name, surname, scholarship, group and averages for the three years of college.

```
(DEFSTRUCT student
                    (surname NIL)          (firstname NIL)
                    (scholarship 0)        (group 0)
                    (gpa_year_1 0)         (gpa_year_2 0)
                    (gpa_year_3 0)
)
```

According to this definition, we evaluate the following forms:

(SETF student1 (make-student))

(SETF student2 (make-student))

(SETF (student-surname student1) 'Ionescu)

(SETF (student-firstname student1) 'Mihai)

(SETF (student-scholarship student1) 1000)

(list (student-firstname student1) (student-surname student1))

9. Arrays

An array is a special type of object in LISP. The arrays are created using the make-array function. To create an array is necessary to specify its size. In some implementations, CLISP and GCLISP including, it is possible to define arrays with one or more dimensions.

The make-array function returns an array. If we want to create an array variable, we will use the construction:

(SETF array (MAKE-ARRAY list-of-sizes))

The array will hold values indexed from **zero** to **size-1**. A few examples follow:

```
(SETF array (MAKE-ARRAY '(10)))
(SETF array (MAKE-ARRAY '(3 3)))
```

An optional list holding initial values may be provided. For example,

```
(SETF array (MAKE-ARRAY '(5) '(1 2 3 4 5) ))
(SETF array (MAKE-ARRAY '(3 3) '((0 1 2) (3 4 5) (6 7 8)) ))
```

We use the item on a particular position in an array as follows:

(AREF array index1 index2 …)

This produces the value contained in the vector [index].

Setting a value to a certain position in an array is done as follows:

(SETF (AREF array index1 index2 …) value)

Specifically, the function AREF returns **value**, and the side effect is the actual assignment **array[index] := value**.

Example

Consider the following array:

```
(SETF x (make-array '(11)))
```

Suppose we would like to initialize the array x with the identical permutation, i.e. the following PSEUDOCODE sequence:

```
for i: = 1, 10 execute
        x[i] := i;
end_for
```

The LISP equivalent of this sequence would be:

```
(DO
        ((i 1 (+ i 1)))
        ((> i 10) nil)
        (SETF (AREF x i) i)
)
```

We mention that in LISP the elements of an array are not necessary to have the same type, in other words arrays with heterogeneous elements (integers, symbols, lists, etc.) can be manipulated.