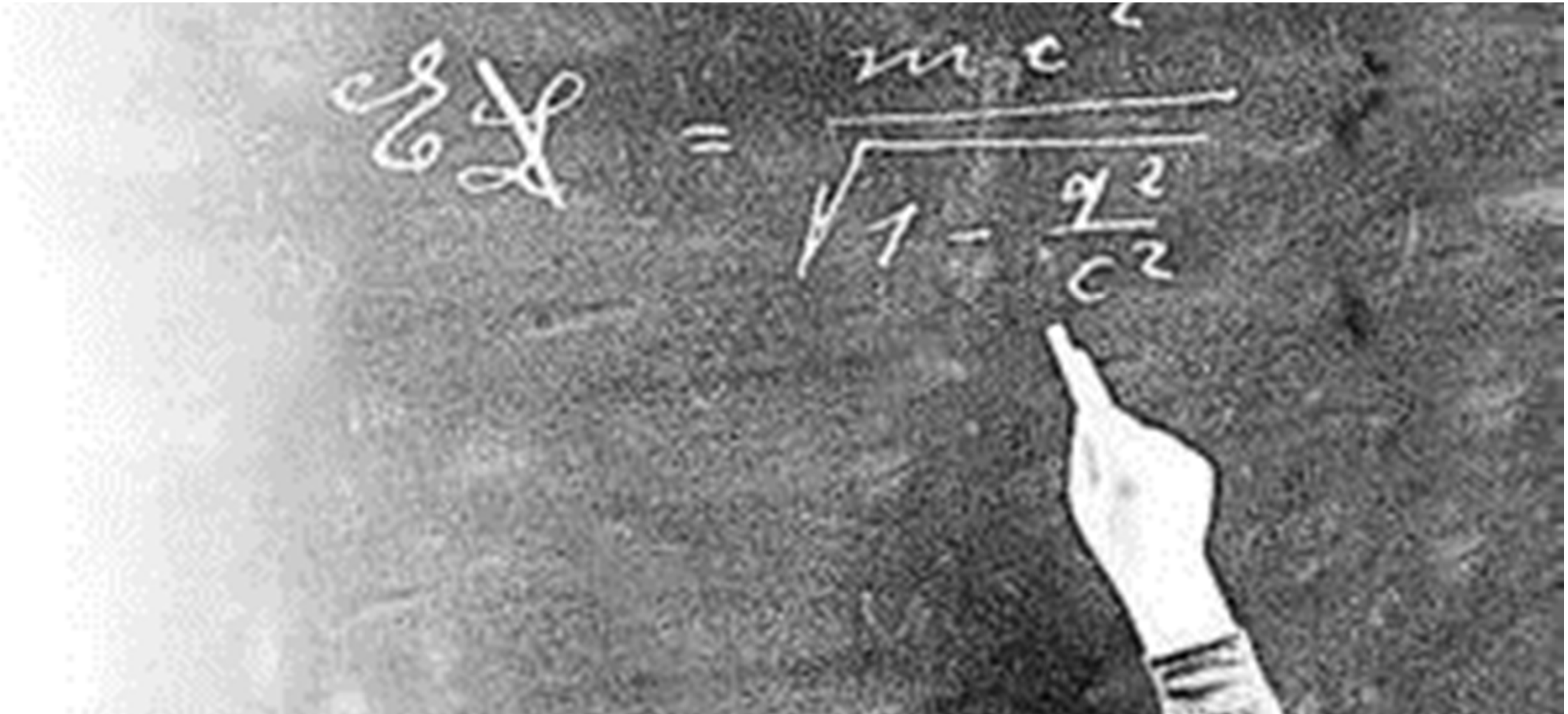


FUNKCIONÁLNÍ A LOGICKÉ PROGRAMOVÁNÍ
5. LISP: STRUKTUROVANÉ DATOVÉ TYPY, FUNKCE JAKO DATA, MAPOVACÍ FUNKCIONÁLY, IMPERATIVNÍ ITERAČN CYKLY V COMMON LISPU

2011 Jan Janoušek
MI-FLP

Evropský sociální fond
Praha & EU:
Investujeme do vaší budoucnosti

BASIC STRUCTURED DATA TYPES

Arrays, vectors

- Simple vectors:

```
>(vector 1 2)
#(1 2)
```

- Vectors and arrays:

`MAKE-ARRAY` creates arrays of any dimensionality.

```
>(make-array 5 :initial-element nil)
#(NIL NIL NIL NIL NIL)
```

Arrays

```
CL-USER 17 > (setf arr1 (make-array '(2 4 6) :initial-
element nil))
#3A(((NIL NIL NIL NIL NIL NIL) (NIL NIL NIL NIL NIL
NIL) (NIL NIL NIL NIL NIL NIL) (NIL NIL NIL NIL NIL
NIL)) ((NIL NIL NIL NIL NIL NIL) (NIL NIL NIL NIL NIL
NIL) (NIL NIL NIL NIL NIL NIL) (NIL NIL NIL NIL NIL
NIL)))
```

- Access to elements – `aref` function

```
CL-USER 26 > (setf (aref arr1 0 2 5)  '(a b c))
(A B C)
CL-USER 27 > (aref arr1 0 2 5)
(A B C)
```

Sequences

- A sequence is a special Lisp type representing **lists, vectors** and **strings.**

Predicate `sequencep`.

Some functions:

```
CL-USER 82 > (length '(1 2 3))
3

CL-USER 86 > (subseq '(1 2 3 4 5 6 7) 1 5)
(2 3 4 5)

CL-USER 90 > (reverse '(1 2 3))
(3 2 1)
2
```

Sequences

- A uniform access to elements: function `elt`

```
CL-USER 97 > (elt "abcd" 1)
#\b

CL-USER 98 > (elt '(1 2 3) 1)
2
```

Some other functions (and their returning values):

`COUNT` Number of times item appears in sequence

`FIND` Item or NIL

`POSITION` Index into sequence or NIL

`REMOVE` Sequence with instances of item removed

`SUBSTITUTE` Sequence with instances of item replaced with new item

`SORT` For sorting sequences

Strings

- String are vectors of characters:

```
CL-USER 55 > (aref "ahoj" 1)
#\h
CL-USER 56 > (char "ahoj" 1)
#\h
```

- Copying strings (sequences in general) is performed by function `copy-seq`.

```
CL-USER 62 > (let ((tmp (copy-seq "polokolo")))
               (setf (char tmp 4) #\p)
               tmp)
"polopolo"
```

Strings

- Concatenation of string (sequences in general) function concatenate.

```
CL-USER 90 > (concatenate 'string "Hello" " "
"World")
"Hello World"
```

- Format function can be used to create a string:

```
CL-USER 91 > (format nil "~A plus ~A je ~A" 3 2
5)
"3 plus 2 je 5"
```

Structures - creating

- Macro `defstruct`

```
CL-USER 8 > (defstruct node
                (value (progn (princ "Zadej cislo: ") (read)))
                (left nil)
               right )
NODE
```

Macro `defstruct` has created:

- `make-node` (creating instances),
- `node-p` (testing type),
- `copy-node` (copying),
- `node-value`, `node-left` and `node-right` (access to the elements of the structure)

Structures

```
CL-USER 9 > (setf nd (make-node))
Zadej cislo: 22
#S(NODE VALUE 22 LEFT NIL RIGHT NIL)

CL-USER 10 > (setf nd2 (make-node :value 33 :left nil :right nil))
#S(NODE VALUE 33 LEFT NIL RIGHT NIL)

CL-USER 11 > (node-p nd)
T
CL-USER 27 > (typep nd 'node)
T
CL-USER 12 > (atom nd)
T
CL-USER 13 > (structurep nd)
T

CL-USER 35 > (equalp nd (copy-node nd))
T
CL-USER 36 > (equal nd (copy-node nd))
NIL

CL-USER 40 > (node-value nd)
22
CL-USER 41 > (setf (node-value nd) 50)
50
CL-USER 42 > nd
#S(NODE VALUE 50 LEFT NIL RIGHT NIL)
```

Structures

Print function can be specified:

```
CL-USER 53 > (defstruct (node (:conc-name nd)
                                   (:print-function (lambda (struct stream depth)
                                     (format stream "#uzel <hodnota: ~A, levy: ~A,
pravy: ~A>"
                                        (ndvalue struct) (ndleft struct) (ndright struct)))))
                  value
                  (left nil)
                  (right nil))
NODE

CL-USER 66 >  (setf nd1 (make-node))
#uzel <hodnota: NIL, levy: NIL, pravy: NIL>

CL-USER 67 >  (ndleft nd1)
NIL
```

…and more…

Hash tables

- General-purpose collection in Lisp
- `Make-hash-table` – creating
- `Gethash` – access to elements
- `Maphash` – mapping functional

Example - hash tables

```
(defparameter *h* (make-hash-table))
(gethash 'foo *h*) ==> NIL
(setf (gethash 'foo *h*) 'quux)
(gethash 'foo *h*) ==> QUUX

;; printing all keys and values
(maphash #'(lambda (k v) (format t "~a => ~a~%" k
v)) *h*)
```

Stack

- List can be used as a stack.
- Functions `push` and `pop`.

```
CL-USER 48 > (setf x nil)
NIL
CL-USER 49 > (push 'a x)
(A)
CL-USER 50 > (push '(b) x)
((B) A)
CL-USER 51 > (pop x)
(B)
```

Sets

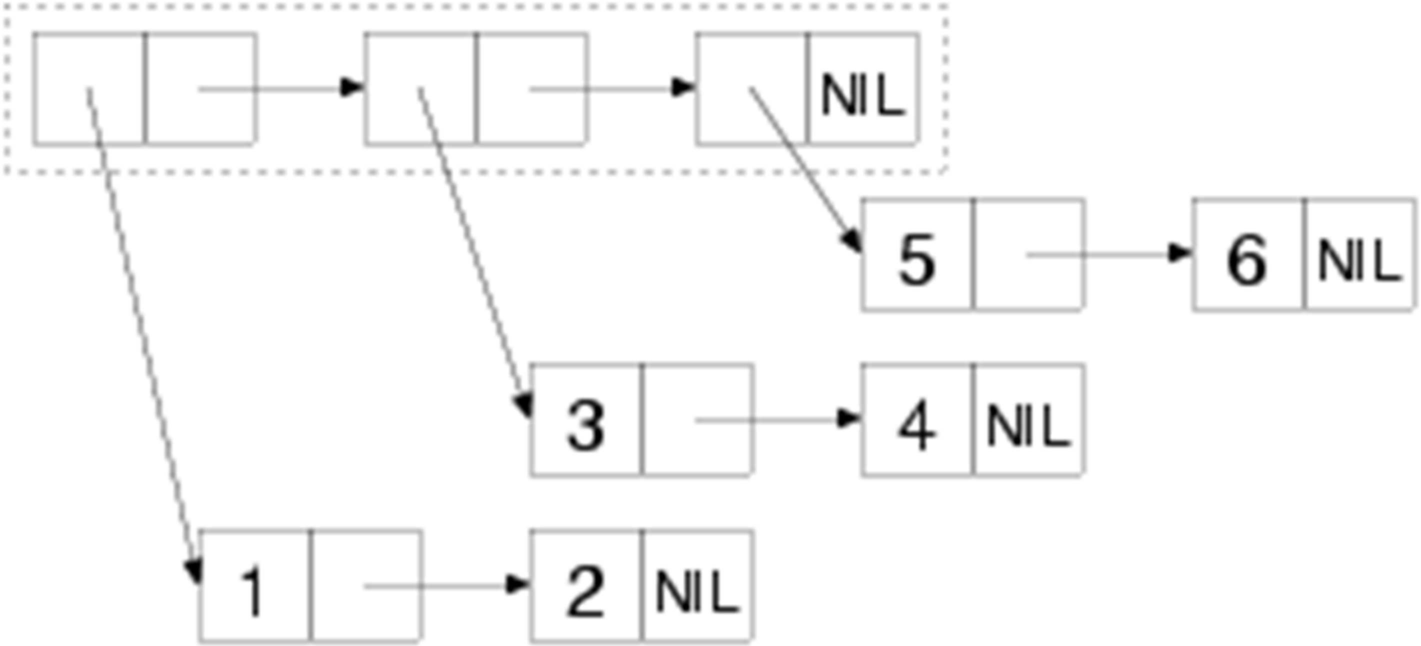
- List can be used as a set.
- Function `member`.

```
CL-USER 24 > (member 3 '(1 2 3 4 5 6))
(3 4 5 6)

CL-USER 26 > (member '(1 2) '((2 3) (1 2) (5 6)))
NIL
; standard comparison by eql

CL-USER 27 > (member '(1 2) '((2 3) (1 2) (5 6)) :test
#'equal )
((1 2) (5 6))
```

Trees

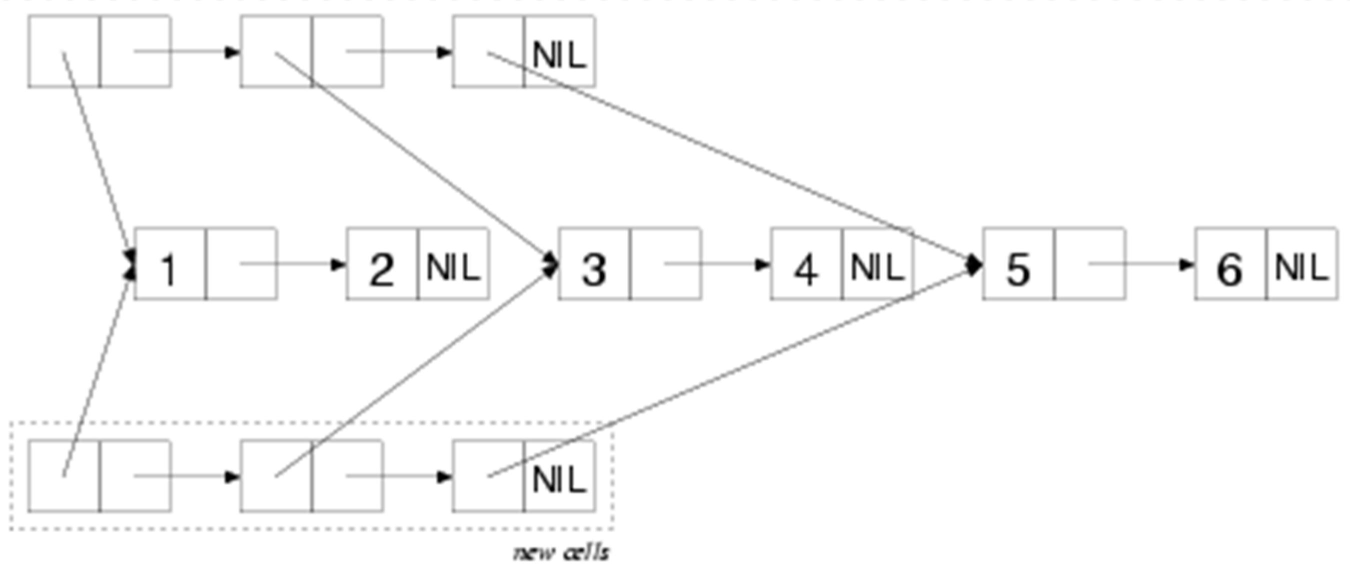
- List can be used as a tree.

((1 2) (3 4) (5 6))



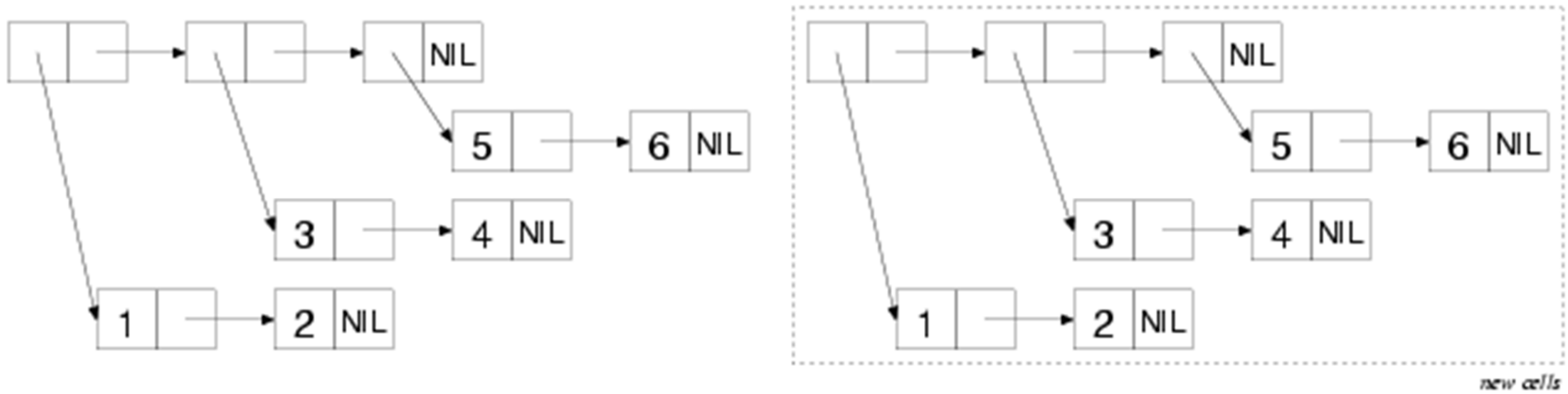
Trees – function copy-tree

Copy-list (does not copy sublists):



Trees – function copy-tree

Copy-tree:



More on functions – functions as data

Functions are treated as data

- can be assigned to a variable
- can be passed as a parameter
- can be return by a function
- can be a part of structured data types

High-order functions – their parameters are function or they return functions. In LISP this is a standard way of programming.

Function operator

- operator `FUNCTION` (aka #) provides the mechanism for getting a function object

```
CL-USER> (defun foo (x) (* 2 x))
FOO

CL-USER> (function foo)
#<Interpreted Function FOO>

CL-USER> #'foo
#<Interpreted Function FOO>
```

Funcall and apply operators

- `FUNCALL` is the one to use when you know the number of arguments you're going to pass to the function:

```
(foo 1 2 3) === (funcall #'foo 1 2 3)
```

- In `APPLY` the second argument after the function object, instead of individual arguments, expects to be a list.

```
(foo 1 2 3) === (apply #'foo `(1 2 3))
```

Example - functions as data

```
(defun plot (fn min max step)
  (loop for i from min to max by step do
        (loop repeat (funcall fn i) do (format t "*"))
        (format t "~%")))
```

```
CL-USER> (plot #'exp 0 4 1/2)
*
*
**
****
*******
************
********************
*********************************
******************************************************
NIL
```

Example - functions as data

```
(defun plot (fn min max step)
  (loop for i from min to max by step do
        (loop repeat (funcall fn i) do (format t "*"))
        (format t "~%")))
```

```
CL-USER> (plot #'exp 0 4 1/2)
*
*
**
****
*******
************
********************
*********************************
******************************************************
NIL
```

Example - functions as data

```
> (sort '(1 4 2 5 6 7 3) #'<)
(1 2 3 4 5 6 7)
NIL
```

Scope – LISP is lexically scoped

Lexical scope (not dynamical)

```
(let ((y 7))
      (defun scope-test (x)
      (list x y)))
```

```
> (let ((y 5))
      (scope-test 3))
(3 7)
```

The result is not: (3 5)

Functions returning functions

Example

```
(defun joiner (obj)
      (typecase obj
            (cons #'append)
            (number #'+)))
```

In this case, only two cases of functions can be written. More can be done with MACROS (see next lecture).

Anonymous functions

Anonymous (unnamed) functions

- General syntax:

```
(lambda (parameters) body)
```

```
(funcall #'(lambda (x y) (+ x y)) 2 3) ==> 5
```

- useful when one needs to pass a function as an argument to another function and the function you need to pass is simple enough to express inline

Anonymous functions - example

```
CL-USER> (plot #'double 0 10 1)

**
****
******
********
**********
************
**************
****************
******************
********************
NIL

CL-USER> (plot #'(lambda (x) (* 2 x)) 0 10 1)

**
****
******
********
**********
************
**************
****************
******************
********************
NIL
```

Iterations – mapping functionals, macros-iteration cycles

Mapping Functionals

Functional = a function with functions in arguments.

```
(defun sum-f (FF S)
  (cond ((null S) 0)
        ((+ (funcall FF (car S))
            (sum-f FF (cdr S)) ))))

(sum-f #'square '(1 2 3))              ; --> 14
```

To apply the same function to all elements of a list:

```
(defun transform (FF S)
  (cond ((null S) NIL)
        ((cons (funcall FF (car S))
               (transform FF (cdr S)) ))))

(transform #'square '(1 2 3))          ; --> (1 4 9)
```

Very common transformation of one list (or more)

$\Rightarrow$ **standard mapping functionals** are in Lisp!

```
(mapcar #'square '(1 2 3))                    ; (1 4 9)
(mapcar #'cons '(1 2 3) '((A) (B) (C)))
(mapcar #'list '(1 2) '(A B) '(X Y))
```

A common format for standard mapping functionals:

(`mapcar` *function list-1 list-2* ... *list-n* **)**

- **`mapcar`**/**`mapc`** operate on successive elements of the lists. The iteration terminates when the shortest *list* runs out, and excess elements in other lists are ignored. The value returned by **`mapcar`** is a list of the results of successive calls to *function*. The value of **`mapc`** is *list-1* .
- **`maplist`**/**`mapl`** are similar to **`mapcar`**/**`mapc`** but operate on sublists, **`maplist`** returns a list of the results, **`mapl`** just *list-1* .
- **`mapcan`**/**`mapcon`** are similar to **`mapcar`**/**`maplist`** but the results are combined into a list as by **`nconc`**

Mapping Functionals – examples

```
CL-USER 1 > (mapcar #'list '(1 2 3) '(4 5 6 7) '(9 8 1 2 3 4))
((1 4 9) (2 5 8) (3 6 1))

CL-USER 2 > (mapcan #'list '(1 2 3) '(4 5 6 7) '(9 8 1 2 3 4))
(1 4 9 2 5 8 3 6 1)

CL-USER 3 > (mapc #'list '(1 2 3) '(4 5 6 7) '(9 8 1 2 3 4))
(1 2 3)

CL-USER 4 > (maplist #'list '(1 2 3) '(4 5 6 7) '(9 8 1 2 3 4))
(((1 2 3) (4 5 6 7) (9 8 1 2 3 4)) ((2 3) (5 6 7) (8 1 2 3 4))
   ((3) (6 7) (1 2 3 4)))
```

Mapping Functionals – another example

Maximal element using `mapc`

```
(defun MyMax (S)
  (let ((MaxEl (car S)))
       (mapc #'(lambda (X)
                     (if (> X MaxEl)(setf MaxEl X)))
             S) ; or (cdr S)
        MaxEl ))
```

Note. `Mapc` is used but for its "side" effects – its returning value is not interesting in this case. Note. Pure functional style does not use any side effects!

Imperative iterations - dolist

- Dolist:

```
(dolist (var list-form)
  body-form*)

CL-USER> (dolist (x '(1 2 3)) (print x))
1
2
3
NIL

CL-USER> (dolist (x '(1 2 3)) (print x) (if (> x 1)
(return)))
1
2
NIL
```

Iterations - dotimes

- Dotimes:

```
(dotimes (var count-form)
  body-form*)

CL-USER> (dotimes (i 4) (print i))
0
1
2
3
NIL
```

Iterations - do

- Do:

```
(do (variable-definition*)
    (end-test-form result-form*)
  statement*)

(var init-form step-form)

(do ((i 0 (1+ i)))
    ((>= i 4))
  (print i))
```

Iterations - loop

- `loop:`

```
(loop
  body-form*)
```

Basic unfinite loop:

```
(loop
  (when (> (get-universal-time) *some-future-date*)
    (return))
  (format t "Waiting~%")
  (sleep 60))
```

Many other variants, for example:

```
(loop for i from 1 to 10 collecting i) ==> (1 2 3 4 5 6 7 8 9 10)
```

An example for as a conclusion: function cc-list in five different ways

Let’s define a function `cc-list` that does the same thing as `copy-list`

```
1. (defun cc-list (list)
     (let ((result nil))
       (dolist (item list result)
         (setf result
          (append result (list item))))))
```

1st implementation uses `append` to put elements onto the end of the list. It traverses the entire partial list each time ⇒ quadratic running time.

An example for as a conclusion: function cc-list in five different ways

2.
```
(defun cc-list (list)
  (let ((result nil))
    (dolist (item list
      (nreverse result))
     (push item result))))
```

2nd implementation goes through the list twice: first to build up the list in reverse order, and then to reverse it. It has linear running time.

3.
```
(defun cc-list (list)
     (let ((result (make-list (length list))))
        (do ((original list (cdr original))
             (new result (cdr new)))
             ((null original) result)
             (setf (car new) (car original)))))
```

4.
```
(defun cc-list (list)
   (mapcar #`identity list))
```

5.
```
(defun cc-list (list)
   (loop for x in list
         collect x))
```

3rd, 4th, and 5th implementations: efficiency usually similar to the 2nd one, depending on the Lisp implementation.

The 4th and 5th implementations are the easiest to understand.