# Robot Programming with Lisp

## 4. More Functional Programming: Map/Reduce, Recursions

Gayane Kazhoyan

Institute for Artificial Intelligence
Universität Bremen

26th April, 2016

# Outline

Theory

Organizational

# Mapping

**Mapping** in functional programming is the process of *applying a function to all members of a list*, *returning a list of results*.

Supported in most functional programming languages and, in addition

- C++ (STL)
- Java 8+
- Python 1.0+
- C# 3.0+
- JavaScript 1.6+
- PHP 4.0+
- Ruby
- Mathematica
- Matlab
- Perl
- Prolog
- Smalltalk, ...

In some of the languages listed the implementation is limited and not elegant.

# Mapping [2]

`mapcar` is the standard mapping function in Common Lisp.

**mapcar** *function list-1* `&rest` *more-lists* ⇒ *result-list*

Apply *function* to elements of *list-1*. Return list of *function* return values.

## mapcar

```
CL-USER> (mapcar #'abs '(-2 6 -24 4.6 -0.2d0 -1/5))
(2 6 24 4.6 0.2d0 1/5)
CL-USER> (mapcar #'list '(1 2 3 4))
((1) (2) (3) (4))
CL-USER> (mapcar #'second '((1 2 3) (a b c) (10/3 20/3 30/3)))
?
CL-USER> (mapcar #'+ '(1 2 3 4 5) '(10 20 30 40))
?
CL-USER> (mapcar #'cons '(a b c) '(1 2 3))
?
CL-USER> (mapcar (lambda (x) (expt 10 x)) '(2 3 4))
?
```

# Mapping [2]

`mapcar` is the standard mapping function in Common Lisp.

**mapcar** *function list-1* `&rest` *more-lists* ⇒ *result-list*

Apply *function* to elements of *list-1*. Return list of *function* return values.

## mapcar

```
CL-USER> (mapcar #'abs '(-2 6 -24 4.6 -0.2d0 -1/5))
(2 6 24 4.6 0.2d0 1/5)
CL-USER> (mapcar #'list '(1 2 3 4))
((1) (2) (3) (4))
CL-USER> (mapcar #'second '((1 2 3) (a b c) (10/3 20/3 30/3)))
(2 B 20/3)
CL-USER> (mapcar #'+ '(1 2 3 4 5) '(10 20 30 40))
(11 22 33 44)
CL-USER> (mapcar #'cons '(a b c) '(1 2 3))
((A . 1) (B . 2) (C . 3))
CL-USER> (mapcar (lambda (x) (expt 10 x)) '(2 3 4))
(100 1000 10000)
```

# Mapping [3]

`mapc` is mostly used for functions with side effects.

**mapc** *function list-1* `&rest` *more-lists* ⇒ *list-1*

## mapc

```
CL-USER> (mapc #'set '(*a* *b* *c*) '(1 2 3))
(*A* *B* *C*)
CL-USER> *c*
3
CL-USER> (mapc #'format '(t t) '("hello, " "world~%"))
hello, world
(T T)
CL-USER> (mapc (alexandria:curry #'format t) '("hello, " "world~%"))
hello, world
("hello~%" "world~%")
CL-USER> (mapc (alexandria:curry #'format t "~a ") '(1 2 3 4))
1 2 3 4
(1 2 3 4)
CL-USER> (let (temp)
           (mapc (lambda (x) (push x temp)) '(1 2 3))
           temp)
```

# Mapping [4]

`mapcan` combines the results using `nconc` instead of `list`.

**mapcan** *function list-1* `&rest` *more-lists* ⇒ *concatenated-results*

If the results are not lists, the consequences are undefined.

## `nconc` vs `list`

```
CL-USER> (list '(1 2) nil '(3 45) '(4 8) nil)
((1 2) NIL (3 45) (4 8) NIL)
CL-USER> (nconc '(1 2) nil '(3 45) '(4 8) nil)
(1 2 3 45 4 8)
CL-USER> (nconc 1 2 nil 3 45 4 8 nil)
; Evaluation aborted on #<TYPE-ERROR expected-type: LIST datum: 1>.
CL-USER> (let ((first-list (list 1 2 3))
               (second-list (list 4 5)))
           (values (nconc first-list second-list)
                   first-list
                   second-list))
(1 2 3 4 5)
(1 2 3 4 5)
(4 5)
```

# Mapping [4]

`mapcan` combines the results using `nconc` instead of `list`.

**mapcan** *function list-1* `&rest` *more-lists* ⇒ *concatenated-results*

If the results are not lists, the consequences are undefined.

## mapcan

```
CL-USER> (mapcar #'list '(1 2 3))
((1) (2) (3))
CL-USER> (mapcan #'list '(1 2 3))
(1 2 3)
CL-USER> (mapcan #'alexandria:iota '(1 2 3))
(0 0 1 0 1 2)
CL-USER> (mapcan (lambda (x)
                   (when (numberp x)
                     (list x)))
                 '(4 n 1/3 ":)"))
(4 1/3)
```

# Mapping [5]

`maplist`, `mapl` and `mapcon` operate on *sublists* of the input list.

**maplist** *function list-1* `&rest` *more-lists* ⇒ *result-list*

## maplist

```
CL-USER> (maplist #'identity '(1 2 3))
((1 2 3) (2 3) (3))
CL-USER> (maplist (lambda (x)
                    (when (>= (length x) 2)
                      (- (second x) (first x))))
                  '(2 2 3 3 3 2 3 2 3 2 2 3))
                     . . .   .   .     .
                   . .       .   .   . .
                   (0 1 0 0 -1 1 -1 1 -1 0 1 NIL)
                      .       .   .     .
                    .   . .           .
                              .   .   .
CL-USER> (maplist (lambda (a-list) (apply #'* a-list)) '(5 4 3 2 1))
(120 24 6 2 1)
```

# Mapping [5]

`maplist`, `mapl` and `mapcon` operate on *sublists* of the input list.

**mapl** *function list-1* `&rest` *more-lists* ⇒ *list-1*

**mapcon** *function list-1* `&rest` *more-lists* ⇒ *concatenated-results*

## mapl

```
CL-USER> (let (temp)
           (mapl (lambda (x) (push x temp)) '(1 2 3))
           temp)
((3) (2 3) (1 2 3))
```

## mapcon

```
CL-USER> (mapcon #'reverse '(4 3 2 1))
(1 2 3 4 1 2 3 1 2 1)
CL-USER> (mapcon #'identity '(1 2 3 4))
; Evaluation aborted on NIL.
```

# Mapping [6]

`map` is a generalization of `mapcar` for *sequences* (lists and vectors).

**map** *result-type function first-sequence* `&rest` *more-sequences* ⇒ *result*

## map

```
CL-USER> (mapcar #'+ #(1 2 3) #(10 20 30))
The value #(1 2 3) is not of type LIST.
CL-USER> (map 'vector #'+ #(1 2 3) #(10 20 30))
#(11 22 33)
CL-USER> (map 'list #'+ '(1 2 3) '(10 20 30))
(11 22 33)
CL-USER> (map 'list #'identity '(#\h #\e #\l #\l #\o))
(#\h #\e #\l #\l #\o)
CL-USER> (map 'string #'identity '(#\h #\e #\l #\l #\o))
"hello"
```

# Reduction

**reduce** *function sequence* `&key` *key from-end start end initial-value* ⇒ *result*

Uses a binary operation, *function*, to combine the elements of *sequence*.

## reduce

```
CL-USER>  (reduce (lambda (x y) (list x y)) '(1 2 3 4))
(((1 2) 3) 4)
CL-USER> (reduce (lambda (x y) (format t "~a ~a~%" x y)) '(1 2 3 4))
1 2
NIL 3
NIL 4
CL-USER> (reduce #'+ '()) ; ?
CL-USER> (reduce #'cons '(3 2 1 nil))
(((3 . 2) . 1))
CL-USER> (reduce #'cons '(3 2 1) :from-end t :initial-value nil)
(3 2 1)
CL-USER> (reduce #'+ '((1 2) (3 4) (5 6))
                 :key #'first :start 1 :initial-value -10)
-2 ; = -10 + 3 + 5
```

# MapReduce

Google's *MapReduce* is a programming paradigm used mostly in huge databases for distributed processing. It was originally used for updating the index of the WWW in their search engine.

Currently supported by AWS, MongoDB, ...

Inspired by the `map` and `reduce` paradigms of functional programming.

`https://en.wikipedia.org/wiki/MapReduce`

# MapReduce [2]
## Example

**Task**: calculate at which time interval the number of travelers on the tram is the highest (intervals are "early morning", "late morning", ...)
**Database**: per interval hourly entries on number of travelers (e.g. db_early_morning: 6:00 → Tram6 → 100, 7:00 → Tram8 → 120)
**Map step**: per DB, go through tram lines and sum up travelers:

- *DB1 early morning:* (Tram6 → 2000) (Tram8 → 1000) ...
- *DB6 late night:* (Tram6 → 200) (Tram4 → 500) ...

**Reduce**: calculate maximum of all databases for each tram line:
Tram6 → 3000 (late morning)
Tram8 → 1300 (early evening)
...

# Local Function Definitions

## flet

```
CL-USER> (defun some-pseudo-code ()
           (flet ((do-something (arg-1)
                    (format t "doing something ~a now...~%" arg-1)))
             (format t "hello.~%")
             (do-something "nice")
             (format t "hello once again.~%")
             (do-something "evil")))
SOME-PSEUDO-CODE
CL-USER> (some-pseudo-code)
hello.
doing something nice now...
hello once again.
doing something evil now...
NIL
CL-USER> (do-something)
; Evaluation aborted on #<UNDEFINED-FUNCTION DO-SOMETHING {101C7A9213}>.
```

# Local Function Definitions [2]

## flet, labels

```
CL-USER> (let* ((lexical-var 304)
                (some-lambda (lambda () (+ lexical-var 100))))
           (let ((lexical-var 4))
             (funcall some-lambda)))
; ?
CL-USER> (let ((lexical-var 304))
           (flet ((some-function () (+ lexical-var 100)))
             (let ((lexical-var 4))
               (some-function))))
; ?
```

# Local Function Definitions [2]

## flet, labels

```
CL-USER> (let* ((lexical-var 304)
                (some-lambda (lambda () (+ lexical-var 100))))
           (let ((lexical-var 4))
             (funcall some-lambda)))
404
CL-USER> (let ((lexical-var 304))
           (flet ((some-function () (+ lexical-var 100)))
             (let ((lexical-var 4))
               (some-function))))
404
CL-USER> (labels ((call-me () (format t "inside CALL-ME~%"))
                  (calling ()
                    (format t "inside CALLING~%")
                    (call-me)))
           (calling))
inside CALLING
inside CALL-ME
```

# Recursion

## Primitive Example

```
CL-USER> (defun dummy-recursion (my-list)
           (when my-list
             (dummy-recursion (rest my-list))))
DUMMY-RECURSION
CL-USER> (trace dummy-recursion)
         (dummy-recursion '(1 2 3 4 5))
  0: (DUMMY-RECURSION (1 2 3 4 5))
    1: (DUMMY-RECURSION (2 3 4 5))
      2: (DUMMY-RECURSION (3 4 5))
        3: (DUMMY-RECURSION (4 5))
          4: (DUMMY-RECURSION (5))
            5: (DUMMY-RECURSION NIL)
            5: DUMMY-RECURSION returned NIL
          4: DUMMY-RECURSION returned NIL
        3: DUMMY-RECURSION returned NIL
      2: DUMMY-RECURSION returned NIL
    1: DUMMY-RECURSION returned NIL
  0: DUMMY-RECURSION returned NIL
```

# Recursion [2]

## Primitive Example #2

```
CL-USER> (defun print-list (list)
           (format t "Inside (print-list ~a)... " list)
           (when list
             (format t "~a~%" (first list))
             (print-list (rest list))))
PRINT-LIST
CL-USER> (print-list '(1 2 3))
Inside (print-list (1 2 3))... 1
Inside (print-list (2 3))... 2
Inside (print-list (3))... 3
Inside (print-list NIL)...
CL-USER> (mapl (lambda (list)
                 (format t "List: ~a... ~a~%" list (first list)))
               '(1 2 3))
List: (1 2 3)... 1
List: (2 3)... 2
List: (3)... 3
(1 2 3)
```

Theory

Organizational

# Recursion [3]

## Length of a List

```
CL-USER> (defun my-length (a-list)
           (if (null a-list)
               0
               (+ 1 (my-length (rest a-list)))))
MY-LENGTH
CL-USER> (trace my-length)
         (my-length '(5 a 3 8))
  0: (MY-LENGTH (5 A 3 8))
    1: (MY-LENGTH (A 3 8))
      2: (MY-LENGTH (3 8))
        3: (MY-LENGTH (8))
          4: (MY-LENGTH NIL)
          4: MY-LENGTH returned 0
        3: MY-LENGTH returned 1
      2: MY-LENGTH returned 2
    1: MY-LENGTH returned 3
  0: MY-LENGTH returned 4
4
```

# Recursion [4]

## Tail Recursion Optimization

```
CL-USER> (defun my-length-inner (a-list accumulator)
           (if (null a-list)
               accumulator
               (my-length-inner (rest a-list) (1+ accumulator))))
MY-LENGTH-INNER
CL-USER> (my-length-inner '(5 a 3 8) 0)
4
CL-USER> (defun my-length-optimal (a-list)
           (my-length-inner a-list 0))
MY-LENGTH-OPTIMAL
CL-USER> (trace my-length-inner)
(MY-LENGTH-INNER)
CL-USER> (my-length-optimal '(5 a 3 8))
...
CL-USER> (untrace my-length my-length-inner)
T
```

# Recursion [5]

## Tail Recursion Optimization: Second Try

```
CL-USER> ,
restart-inferior-lisp
CL-USER> (proclaim '(optimize speed))
; No value
CL-USER> (defun my-length-inner (a-list accumulator)
           (if (null a-list)
               accumulator
               (my-length-inner (rest a-list) (1+ accumulator))))
CL-USER> (defun my-length-optimal (a-list)
           (my-length-inner a-list 0))
CL-USER> (trace my-length-optimal my-length-inner)
(MY-LENGTH-OPTIMAL MY-LENGTH-INNER)
CL-USER> (my-length-optimal '(5 a 3 8))
  0: (MY-LENGTH-OPTIMAL (5 A 3 8))
    1: (MY-LENGTH-INNER (5 A 3 8) 0)
    1: MY-LENGTH-INNER returned 4
  0: MY-LENGTH-OPTIMAL returned 4
4
```

# Recursion [6]

## What Does This Function Do?

```
CL-USER> (defun sigma (n)
           (labels ((sig (c n)
                      (declare (type fixnum n c))
                      (if (zerop n)
                          c
                          (sig (the fixnum (+ n c))
                               (the fixnum (- n 1))))))
             (sig 0 n)))
SIGMA
CL-USER> (trace sigma)
(SIGMA)
CL-USER> (sigma 5)
  0: (SIGMA 5)
  0: SIGMA returned 15
15
```

```
(declare (type typespec var*)
(the return-value-type form)
```

# Links

- Functional programmer Bible (available for download):

  `http://www.paulgraham.com/onlisp.html`

# Outline

Theory

Organizational

# Info Summary

- Assignment code: `REPO/assignment_4/src/*.lisp`
- Assignment due: 03.05?, Tuesday, 08:00 AM German time
- Next class: 03.05, 16:15

# Q & A

Thanks for your attention!