Robot Programming with Lisp

4. More Functional Programming: Map/Reduce, Recursions

Gayane Kazhoyan

Institute for Artificial Intelligence
Universität Bremen

4th November, 2014

Outline

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 also

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

In some languages the implementation is ugly and limited.

Mapping [2]

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

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

```
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> (setf temp nil)
         (mapc #'(lambda (x) (push x temp)) '(1 2 3))
         temp
```

Mapping [4]

`mapcan` combines the results using `nconc` instead of `list`.
If the results are not lists, the consequences are undefined.

nconc

```
CL-USER> (setf *first-list* (list 1 2 3) *second-list* (list 4 5))
         (nconc *first-list* *second-list*)
(1 2 3 4 5)
CL-USER> *first-list*
(1 2 3 4 5)
CL-USER> (nconc '(1 2) nil '(3 45) '(4 8) nil)
(1 2 3 45 4 8)
```

mapcan

```
CL-USER> (mapcar #'list '(1 2 3))
((1) (2) (3))
CL-USER> (mapcan #'list '(1 2 3))
(1 2 3)
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 successive sublists of the input list, as opposed to single elements thereof.

maplist

```
CL-USER> (maplist #'identity '(1 2 3))
((1 2 3) (2 3) (3))
CL-USER> (maplist (lambda (x) (unless (< (length x) 2)
                                (cond ((< (first x) (second x)) 1)
                                      ((> (first x) (second x)) -1)
                                      (t 0))))
                  '(0 0 1 1 1 0 1 0 1 0 0 1))
(0 1 0 0 -1 1 -1 1 -1 0 1 NIL)

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

Mapping [6]

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

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 '(#\1 #\2 #\3))
(#\1 #\2 #\3)
CL-USER> (map 'string #'identity '(#\1 #\2 #\3))
"123"
```

Reduction

reduce *function sequence &key key from-end start end initial-value => result*
Description: `reduce` uses a binary operation, `function`, to combine the elements of `sequence` bounded by `start` and `end`.

reduce

```
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 #'+ #(1 2 3))
6
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 ; = 3 + 5 - 10
```

MapReduce

Application of `map` and `reduce` outside of functional programming.

Google's "invented" *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, ...

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 "nasty")))
SOME-PSEUDO-CODE
CL-USER> (some-pseudo-code)
hello...
doing something nice now...
hello once again...
doing something nasty now...
NIL
CL-USER> (do-something)
The function COMMON-LISP-USER::DO-SOMETHING is undefined.
```

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))))
; ?
CL-USER> (labels ((call-me () (format t "inside CALL-ME~%"))
                  (i-call-you ()
                    (format t "inside I-CALL-YOU~%")
                    (call-me)))
           (i-call-you))
inside I-CALL-YOU
inside CALL-ME
```

Recursion

Primitive Example

```
CL-USER> (defun print-list (list)
           (format t "Inside (print-list ~a)... " list)
           (unless (null 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 "The list is ~a... ~a~%" list (first list)))
               '(1 2 3))
The list is (1 2 3)... 1
The list is (2 3)... 2
The list is (3)... 3
```

Recursion [2]

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 [3]

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 (a-list)
           (my-length-inner a-list 0))
STYLE-WARNING: redefining COMMON-LISP-USER::MY-LENGTH in DEFUN
MY-LENGTH
CL-USER> (trace my-length-inner)
(MY-LENGTH-INNER)
CL-USER> (my-length '(5 a 3 8))
...
CL-USER> (untrace my-length my-length-inner)
T
```

Recursion [4]

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 (a-list)
           (my-length-inner a-list 0))
CL-USER> (trace my-length my-length-inner)
(MY-LENGTH MY-LENGTH-INNER)
CL-USER> (my-length '(5 a 3 8))
  0: (MY-LENGTH (5 A 3 8))
    1: (MY-LENGTH-INNER (5 A 3 8) 0)
    1: MY-LENGTH-INNER returned 4
  0: MY-LENGTH returned 4
```

Recursion [5]

What Does This Function Do?

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

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

Links

- Pro lisper / functional programmer Bible (available for download):
  `http://www.paulgraham.com/onlisp.html`

Outline

Info Summary

- Assignment code: `REPO/assignment_4/src/*.lisp`
- Assignment due: 09.11, Sunday, 23:59 German time
- Assignment solutions: discussed in the class
- Next class: 11.11, 14:15, room TAB 1.58 (1. OG)
- Lecturer: Gayane Kazhoyan

Q & A

Thanks for your attention!