

Московский государственный технический университет
имени Н. Э. Баумана



Факультет: Информатика и системы управления

Кафедра: Программное обеспечение ЭВМ и информационные технологии

Дисциплина: Функциональное и логическое программирование

Лабораторная работа №5

Выполнили: Никичкин А. С., Фокеев А. И.
Группа: ИУ7–61

Москва, 2015 г.

Полезные функции

```
1 (defun pos-element-in-list (element l)
2   "Return mask where the element in the set"
3   (mapcar #'(lambda (x)
4               (if (equal element x) 1 0))
5             l))
6
7 (defun c-count (element l)
8   (let ((mask (pos-element-in-list element l)))
9     (reduce #'+ mask)))
10
11 (defun in-list (element l)
12   "Return T if the element in the list"
13   (let* ((bits (pos-element-in-list element l))
14          (i (c-count 1 bits)))
15     (declare (fixnum i))
16     (< 0 i)))
17
18 (defun not-in-list (element l)
19   "Return T if the element not in the list"
20   (not (in-list element l)))
21
22 (defun symbol-or-listp (arg)
23   (or (symbolp arg)
24       (listp arg)))
25
26 (defun c-list-length (l)
27   (let ((bits (mapcar #'symbol-or-listp l)))
28     (c-count t bits)))
29
30 (defun _c-reverse (l)
31   (let ((buf (list nil)))
32     (mapcar #'(lambda (_element)
33                 (push _element (first buf)))
34             l)
35     (car buf)))
36
37 (defun c-reverse (l)
38   (let ((len (c-list-length l)))
39     (cond
40      ((< len 2) l)
41      (t (_c-reverse l)))))
```

- 1 Функция, которая по своему списку-аргументу lst определяет является ли он палиндромом (то есть равны ли lst и '(reverse lst))

```
1 (defun palindromep (l)
2   (equal l
```

```

3      (c-reverse l)))
4
5 (defun is-palindromes (l)
6   (mapcar #'palindromep l))

```

2 Предикат `set-equal`, который возвращает Т, если два его множества-аргумента содержат одни и те же элементы, порядок которых не имеет значения

```

1 (defun _normalize-set (set)
2   "Return new normalized set"
3   (let ((result-set '(), (first set))))
4     (mapcar #'(lambda (_element)
5                 (if (not-in-list _element result-set)
6                     (rplacd (last result-set) '(_element))))
7           set)
8   result-set))
9
10 (defun normalize-set (set)
11   "Some condition before run normalize-set"
12   (cond
13     ((null set) nil)
14     (t (_normalize-set set))))
15
16 (defun mask-elements-in-set (set1 set2)
17   "Return mask of positions"
18   (mapcar #'(lambda (_element)
19               (if (in-list _element set2) 1 0))
20         set1))
21
22 (defun _set-equalp (set1 set2)
23   "Internal for set-equalp without validation"
24   (let ((mask (mask-elements-in-set set1 set2)))
25     (eql (c-count 1 mask) (c-list-length set2))))
26
27 (defun set-equalp (set1 set2)
28   "Test on equal of sets"
29   (let* ((normal-set1 (normalize-set set1))
30          (normal-set2 (normalize-set set2))
31          (length-set1 (c-list-length normal-set1))
32          (length-set2 (c-list-length normal-set2)))
33     (declare (fixnum length-set1 length-set2))
34     (cond
35       ((/= length-set1 length-set2) nil)
36       (t (_set-equalp normal-set1 normal-set2)))))

```

3 Функции, которые обрабатывают таблицу из точечных пар (страна . столица) и возвращают по стране — столицу, а по столице — страну

```
1 (setf test-table '((country1 . city1)
2                    (country2 . city2)
3                    (country3 . city3)
4                    (country4 . city4)
5                    (country5 . city5)))
6
7 (defun get-city (country table)
8   (rest (find-if #'(lambda (x)
9                     (equal (first x) country))
10          table)))
11
12 (defun get-country (city table)
13   (first (find-if #'(lambda (x)
14                     (equal (rest x) city))
15          table)))
```

4 Функция, которая переставляет в списке-аргументе первый и последний элемент

4.1 с использованием rplaca и rplacd

```
1 (defun swap-first-last-dl1 (dl)
2   "Swap for dotted-list"
3   (let ((t-first (car dl))
4         (t-last (last dl 0)))
5     (cond
6       ((consp dl) (rplaca dl t-last)
7                   (rplacd (last dl) t-first)
8                   dl)
9       (t nil))))
10
11 (defun swap-first-last-l1 (l)
12   "Swap for list"
13   (let ((t-first (car l))
14         (t-last (car (last l))))
15     (cond
16       ((consp l) (rplaca l t-last)
17                  (rplacd (last l 2) '(, t-first))
18                  l)
19       (t nil))))
20
21 (defun swap-first-last1 (l)
22   "Smart swap"
23   (cond
24     ((last l 0) (swap-first-last-dl1 l))
25     (t (swap-first-last-l1 l))))
```

4.2 с использованием butlast

```
1 (defun swap-first-last-dl2 (dl)
2   "Swap for dotted-list with copy"
3   (let ((left (first dl))
4         (mid (butlast (rest dl) 0))
5         (right (last dl 0)))
6     (if (consp dl)
7         '(,right ,@mid . ,left)
8         nil)))
9
10 (defun swap-first-last-l2 (l)
11   "Swap for dotted-list with copy"
12   (let ((left (first l))
13         (mid (butlast (rest l)))
14         (right (car (last l))))
15     (if (consp l)
16         '(,right ,@mid ,left)
17         nil)))
18
19 (defun swap-first-last2 (l)
20   "Smart swap"
21   (cond
22     ((not (or (consp (cdr l))
23               (null (cdr l)))))
24     '(', (cdr l) . , (car l)))
25   ((< (length l) 2) l)
26   ((last l 0) (swap-first-last-dl2 l))
27   (T (swap-first-last-l2 l)))
```

4.3 с использованием remove-if

```
1 (defun rm (fn l &key from-end)
2   (funcall fn
3     #'(lambda (x)
4         (or (equal x (car l))
5             (equal x (car (last l)))))
6     l
7     :count 1
8     :from-end from-end))
9
10 (defun swap-first-last3 (l)
11   "Swap but dotted-list"
12   (cond
13     ((< (length l) 2) l)
14     (T '(', (car (last l))
15         ,@(rm #'remove-if (rm #'remove-if l) :from-end T)
16         ,(first l))))
```

5 Функция, которая переставляет в списке-аргументе два указанных своими порядковыми номерами элемента в этом списке

```
1 (defun swap (lst pos1 pos2)
2   "Swap two elements by pos1 and pos2"
3   (declare (integer pos1 pos2))
4   (let ((x (nth pos1 lst))
5         (y (nth pos2 lst))
6         (l (length lst)))
7     (cond ((and (> l 2)
8                 (< 0 pos1 l)
9                 (< 0 pos2 l))
10          (rplaca (nthcdr pos1 lst) y)
11          (rplaca (nthcdr pos2 lst) x)
12          lst)
13     (T (format T "Something wrong")))))
```

6 Функции, которые производят круговую перестановку в списке-аргументе влево и вправо

6.1 циклический сдвиг влево

```
1 (defun cycle-shift-left (l)
2   "Cycle shift list to left"
3   (let ((f (first l)))
4     (maplist #'(lambda (_x)
5                  (rplaca _x (second _x)))
6              l)
7     (rplaca (last l) f)
8     l))
```

6.2 циклический сдвиг вправо

```
1 (defun cycle-shift-right (l)
2   "Cycle shift list to right"
3   (let ((hold nil)
4         (before nil)
5         (last-el (car (last l))))
6     (maplist #'(lambda (_x)
7                  (setf hold (first _x))
8                  (rplaca _x before)
9                  (setf before hold))
10              l)
11     (rplaca l last-el)))
```

7 Функция, которая умножает на заданное число-аргумент все числа из заданного списка-аргумента

7.1 все элементы списка — числа

```
1 (defun multiply-by (l k)
2   "Multiply list of numbers by k"
3   (declare (fixnum k))
4   (mapcar #'(lambda (_x) (* _x k))
5             l))
```

7.2 элементы списка — любые объекты

```
1 (defun multiply-by (l k)
2   "Multiply list of numbers by k"
3   (declare (fixnum k))
4   (mapcar #'(lambda (_x) (* _x k))
5             l))
```

8 Функция, которая из списка-аргумента, содержащего только числа, выбирает только те, которые расположены между двумя указанными границами-аргументами и возвращает их в виде списка упорядоченного по возрастанию списка чисел

```
1 (defun _between (nums l r)
2   (declare (fixnum l r))
3   (mapcar #'(lambda (_x)
4               (if (<= l _x r)
5                   _x))
6           nums))
7
8 (defun between (nums l r)
9   (declare (fixnum l r))
10  (let ((data (_between nums l r)))
11    (remove-if-not #'numberp data)))
12
13 (defun sort-nums-between (nums l r)
14   (declare (fixnum l r))
15   (let ((data (between nums l r)))
16     (sort data #'<)))
```