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

```
(defun pos-element-in-list (element lst)
  "Return mask where the element in the set"
  (mapcar #'(lambda (x) (if (equal element x) 1 0))
          lst))

(defun c-count (element lst)
  (let ((mask (pos-element-in-list element lst)))
    (reduce #'+ mask)))

(defun in-list (element lst)
  "Return T if the element in the list"
  (let* ((bits (pos-element-in-list element lst))
         (i (c-count 1 bits)))
    (declare (fixnum i))
    (< 0 i)))

(defun not-in-list (element lst)
  "Return T if the element not in the list"
  (not (in-list element lst)))

(defun symbol-or-listp (arg)
  (or (symbolp arg)
      (listp arg)))

(defun c-list-length (lst)
  (let ((bits (mapcar #'symbol-or-listp lst)))
    (c-count t bits)))

(defun _c-reverse (l &aux (buf (list nil)))
  (mapcar #'(lambda (_element) (push _element (first buf)))
          l)
  (car buf))

(defun c-reverse (lst &aux (len (c-list-length l)))
  (cond ((< len 2) lst)
        (t (_c-reverse lst))))
```

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

```
(defun palindromep (lst)
  (equal lst
         (c-reverse lst)))

(defun is-palindromes (lst)
  (mapcar #'palindromep lst))
```

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

```
(defun _normalize-set (set)
  "Return new normalized set"
  (let ((result-set '(,(first set))))
    (mapcar #'(lambda (_element)
                (if (not-in-list _element result-set)
                    (rplacd (last result-set) '(,_element))))
            set)
    result-set))

(defun normalize-set (set)
  "Some condition before run normalize-set"
  (cond
    ((null set) nil)
    (t (_normalize-set set))))

(defun mask-elements-in-set (set1 set2)
  "Return mask of positions"
  (mapcar #'(lambda (_element)
                      (if (in-list _element set2) 1 0))
          set1))

(defun _set-equalp (set1 set2)
  "Internal for set-equalp without validation"
  (let ((mask (mask-elements-in-set set1 set2)))
    (eql (c-count 1 mask) (c-list-length set2))))

(defun set-equalp (set1 set2)
  "Test on equal of sets"
  (let* ((normal-set1 (normalize-set set1))
         (normal-set2 (normalize-set set2))
         (length-set1 (c-list-length normal-set1))
         (length-set2 (c-list-length normal-set2)))
    (declare (fixnum length-set1 length-set2))
    (cond ((/= length-set1 length-set2) nil)
          (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

```
(setf test-table '((country1 . city1)
                   (country2 . city2)
                   (country3 . city3)
                   (country4 . city4)
                   (country5 . city5)))

```

```
(defun get-city (country table)
  (rest (find-if #'(lambda (x)
                     (equal (first x) country))
                 table)))

(defun get-country (city table)
  (first (find-if #'(lambda (x)
                      (equal (rest x) city))
                  table)))
```

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

4.1 с использованием `rplaca` и `rplacd`

```
(defun swap-first-last-dl1 (dl)
  "Swap for dotted-list"
  (let ((t-first (car dl))
        (t-last (last dl 0)))
    (cond
      ((consp dl) (rplaca dl t-last)
                  (rplacd (last dl) t-first)
                  dl)
      (t nil))))

(defun swap-first-last-l1 (l)
  "Swap for list"
  (let ((t-first (car l))
        (t-last (car (last l))))
    (cond
      ((consp l) (rplaca l t-last)
                 (rplacd (last l 2) `(,t-first))
                 l)
      (t nil))))

(defun swap-first-last1 (l)
  "Smart swap"
  (cond
    ((last l 0) (swap-first-last-dl1 l))
    (t (swap-first-last-l1 l))))
```

4.2 с использованием `butlast`

```
(defun swap-first-last-dl2 (dl)
  "Swap for dotted-list with copy"
  (let ((left (first dl))
        (mid (butlast (rest dl) 0))
        (right (last dl 0)))
    (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 (lst coef)
2    (let ((item (first lst))
3          (tail (rest lst)))
4      (cond ((null lst) nil)
5            ((numberp item)
```

```
6             (cons (* item coef)
7                   (multiply-by tail coef)))
8           ((listp item)
9             (cons (multiply-by item coef)
10                  (multiply-by tail coef)))
11          (t (format t "Error: ~a is not a number" item)))))
```

7.2 элементы списка — любые объекты

```
1  (defun smart-multiply-by (lst coef)
2    (let ((item (first lst))
3          (tail (rest lst)))
4      (cond ((null lst) nil)
5            ((numberp item)
6             (cons (* item coef)
7                   (smart-multiply-by tail coef)))
8            ((listp item)
9             (cons (smart-multiply-by item coef)
10                  (smart-multiply-by tail coef)))
11           (t
12            (cons item
13                  (smart-multiply-by tail coef))))))
```

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