Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н. Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №8

По курсу: "Функциональное и Логическое программирование"

Тема Использование функционалов.

Группа ИУ7-63Б

Студент Сукочева А.

Преподаватель Толпинская Н.Б.

Преподаватель Строганов Ю. В.

Москва — 2021 г.

Практическая часть

Задание 1. Напишите функцию, которая умножает на заданное число-аргумент все числа из заданного списка-аргумента, когда

а) все элементы списка — числа,

```
(defun f (lst num)
    (cond ((null lst) ())
    (T (cons (* num (car lst)) (f (cdr lst) num))) ) )
```

Пример использования:

```
(f '(1 2 3 4) 5) ;; (5 10 15 20)
```

б) элементы списка — любые объекты.

```
(defun f (lst num)
    (cond ((null lst) ())
    ((symbolp (car lst)) (cons (car lst) (f (cdr lst) num)))
    ((listp (car lst)) (cons (f (car lst) num) (f (cdr lst) num)))
    (T (cons (* num (car lst)) (f (cdr lst) num))) ) )
```

Пример использования:

```
(f '(1 2 (3 4 a) (b) T 7) 2) ;; (2 4 (6 8 A) (B) T 14)
```

Задание 2. Напишите функцию, `select-between`, которая из списка-аргумента, содержащего только числа, выбирает только те, которые расположены между двумя указанными границами-аргументами и возвращает их в виде списка

```
(defun check-border (x a b)
    (and (>= x a) (<= x b)) )

(defun select-between (lst a b)
        (cond ((null lst) ())
        ((symbolp (car lst)) (cons (car lst) (select-between (cdr lst) a b)))
        ((listp (car lst)) (cons (select-between (car lst) a b)
           (select-between (cdr lst) a b)))
        ((check-border (car lst) a b) (cons (car lst) (select-between (cdr
           lst) a b)))
        (T (select-between (cdr lst) a b))) )
```

Пример использования:

```
(select-between '(1 2 (a b 3 4) T c 4 6 11 5) 2 7) ;; (2 (A B 3 4) T C 4 6 5)
```

Задание 3. Что будет результатом (mapcar 'вектор '(570-40-8))

mapcar примерняет свой первый аргумент поэлементно к своим аргументам. Т.е. первым аргументом должна быть функция. В нашем случае функции 'вектор' нет.

Результат: Error: ВЕКТОР is undefined.

Задание 4. Напишите функцию, которая уменьшает на 10 все числа из списка аргумента этой функции.

```
(defun f-func (lst)
    (mapcar (lambda (x) (- x 10)) lst))
```

```
(defun f-rec (lst)
    (cond ((null lst) ())
    ((symbolp (car lst)) (cons (car lst) (f-rec (cdr lst))))
    ((listp (car lst)) (cons (f-rec (car lst)) (f-rec (cdr lst))))
    (T (cons (- (car lst) 10) (f-rec (cdr lst))))) )
```

Пример использования:

```
(f-func '(11 12 13 14 1))        ;; (1 2 3 4 -9)
(f-rec '(11 12 13 14 1))         ;; (1 2 3 4 -9)
(f-rec '(11 12 (14 b 15) 16))    ;;(1 2 (4 B 5) 6)
```

Задание 5. Написать функцию, которая возвращает первый аргумент списка-аргумента, который сам является непустым списком.

```
(defun f (lst)
        (cond ((null lst) NIL)
        ((null (car lst)) NIL)
        (T (car lst))) )
```

Пример использования:

```
(f '(Nil 1 2 3)) ;; NIL
(f '((1 2 3) 4 5 6)) ;; (1 2 3)
```

Задание 6. Сумма числовых элементов смешанного структурированного списка

```
(defun f-rec (lst num)
        (cond ((null lst) num)
        ((symbolp (car lst)) (f-rec (cdr lst) num))
        ((listp (car lst)) (+ (f-rec (car lst) 0) (f-rec (cdr lst) num)))
        (T (f-rec (cdr lst) (+ num (car lst))))) )

(defun f (lst)
        (f-rec lst 0) )
```

Пример использования:

```
(f '(1 2 3 (a b c) (a 2 b) (((c))) ((5)))) ;; => 13
```

Теоретическая часть

Порядок работы и варианты использования функционалов

Функционалы - функции, которые в качестве параметра принимают функцию.

Применяющие функционалы позволяют применить свой функциональный аргумент (функцию) к заданным её аргументам.

Функционал **apply** является функцией с двумя вычисляемыми аргументами, обращение к ней имеет вид:

```
    (apply F L)
```

где F – функциональный аргумент и L – список: $L => (p_1...p_n)$, $n \geq 0$ рассматриваемый как список фактических параметров для F. Значение функционала – результат применения F к этим фактическим параметрам,

Пример:

```
(apply #'* '(2 5))                    ;; => 10
(apply (lambda (x) (- x 10)) '(2))    ;; => -8
```

Функционал **funcall** является специальной функцией. Обращение к ней:

```
(funcall F e1 ... en )
```

Где $n \geq 0$. Её действие аналогично apply, отличие состоит в том, что аргументы применяемой функции F задаются не списком, а по отдельности.

Пример:

```
(funcall #'* '2 5)                    ;; => 10
(funcall (lambda (x) (- x 10)) 2)     ;; => -8
```

В группу отображающих функционалов входят функции mapcar и maplist. Их имена имеют префикс map (mapping – отображение), поскольку их действие – отображение списка-аргумента в список-результат за счёт применения заданной функции к элементам исходных списков.

Обращение к **mapcar** имеет вид:

```
(mapcar F L1 L2 ... Ln)
```

Функционал mapcar последовательно применяет свой функциональный аргумент F (функцию от одного аргумента) к элементам списков L_i, которые извлекаются из L_i функцией-селектором car (поэтому имя car входит в имя функционала), и возвращает список из полученных значений.

Схематично результат может быть записан как:

```
((F (car L1) ... (car Ln)) (F (cadr L1) ... (cadr Ln)) (F (caddr L1) ...
   (caddr Ln)) ...).
```

Пример:

```
(mapcar #'length '((A B) (C) (D E F)))   ;; =>(2 1 3)
(mapcar #'list '(A S D F))                ;; => ((A)(S)(D)(F))
(mapcar (lambda (x) (if (< x 0) (* -1 x) x))   '(-1 2 3 -4 5)) ;; => (1 2 3 4 5)
(mapcar #'* '(1 2 3) '(4 5 6))            ;; => (4 10 18)
```

Обращение к **maplist** имеет вид:

```
(maplist F L1 ... Ln)
```

Функционал **maplist** последовательно применяет свой функциональный аргумент F к спискам-аргументам L_i и их хвостовым частям (полученным из L путём отбрасывания первого элемента, первых двух элементов и т.д.) и возвращает список из вычисленных значений.

Схематично действие этого функционала можно записать как

```
((F L1 ... Ln) (F (cdr L1) ... (cdr Ln)) (F (cddr L1) ... (cddr Ln))...)
```

Пример:

```
(maplist #'list '(a b c d))
;; => (((A B C D)) ((B C D)) ((C D)) ((D)))

(maplist #'list '(1 2 3) '(4 5 6))
;; => (((1 2 3) (4 5 6)) ((2 3) (5 6)) ((3) (6)))
```

```
(find-if F L)
```

find-if применяет предикат F к списку аргументу L, и возвращает первый элемент, для которого результат применяемого предиката отличен от Nil. В случае, если все элементы не удовлетворяют предикату возвращается Nil:

Пример:

```
(find-if #'evenp '(1 2 3 4 5))  ;; => 2
(find-if #'evenp '(1 3 5))      ;; => Nil
(find-if (lambda (x) (> x 0)) '(-5 1 2 5 -1)) ;; 1
```

```
(remove-if F L)
```

remove-if (**remove-if-not**) возвращает список без элементов для которых истин предикат F.

Пример:

```
(remove-if #'evenp '(1 2 3))         ;; => (1 3)
(remove-if-not #'evenp '(1 2 3))     ;; => (2)
```

every возвращает Nil, как только предикат (первый аргумент) для очередного аргумента списка-аргумента вернул Nil, иначе, если применение предиката для каждого элемента вернуло T, вернется T.

Пример:

```
(every #'symbolp '(a b c d))     ;; T
(every #'symbolp '(a b c 1))     ;; Nil

(every #'numberp '(1 2 3))       ;; T
(every #'numberp '(1 2 a))       ;; Nil
```

some возвращает истину, как только предикат (первый аргумент) для очередного аргумента списка-аргумента вернул истину, иначе, если применение предиката для каждого элемента вернуло Nil, вернется Nil.

```
(some #'numberp '(1 2 3)) ;; => T
(some #'numberp '(1 2 a)) ;; => T
(some #'numberp '(a b c)) ;; => Nil
```

reduce реализует редукцию заданного списка. Применяет функцию F к первому элементу и начальному значению A, далее результат служит аргументом для применения этой же функции и второго аргумента и т.д..

Схематично можно представить так:

```
  (reduce F L :initial-value A) = (F(...(F(F A e 1 ) e 2 ))...e n )
```

Пример:

```
(reduce #'* '(1 2 3))                                    ;; => 6
(reduce (lambda (x y) (+ x y)) '(1 2 3))                 ;; => 6
(reduce (lambda (x y) (+ x y)) '(1 2 3) :initial-value 10)  ;; => 16
```