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»

Лабораторная работа №9
Использование функционалов и рекурсии

Студент: Луговой Д.М.
Группа: ИУ7-61Б
Преподаватель: Толпинская Н.Б.

Москва, 2020 г.

Цель работы: приобрести навыки использования функционалов и рекурсии.

Задачи работы: изучить работу и методы использования отображающих функционалов: mapcar, maplist, reduce и др., изучить способы организации хвостовой рекурсии, сравнить эффективность.

Лабораторная работа 5

Задание 2

Написать предикат set-equal, который возвращает t, если два его множества-аргумента содержат одни и те же элементы, порядок которых не имеет значения.

Реализация с помощью функционалов:

Листинг 1: Функция проверки эквивалентности двух множеств

```
(defun check-sets-equal (set1 set2)
  (cond
    (
      (and
        (reduce #'(lambda (x y) (and x y))
          (mapcar #'(lambda (x)
              (mapcan #'(lambda (y)
                  (cond
                    (
                      (equal x y)
                      (cons y nil)
                    )
                  )
                ) set2
              )
            ) set1
          )
        )
        (reduce #'(lambda (x y) (and x y))
          (mapcar #'(lambda (x)
              (mapcan #'(lambda (y)
                  (cond
                    (
                      (equal x y)
                      (cons y nil)
                    )
                  )
                ) set1
              )
            ) set2
```

```
31              )
32            )
33          )
34          T
35        )
36      )
37  )
```

set1 и **set2** - списки-множества.

С помощью mapcan идет проверка на вхождение элемента во множество, возвращается список из элементов другого множества, равных текущему элементу, или nil, если таких нет. С помощью mapcar определяется факт вхождения каждого из элементов одного множества в другое. Затем с помощью reduce осуществляется проверка на то, что каждый элемент текущего множества есть в другом множестве. Проверку на то, что множество является подмножеством другого множества необходимо произвести для каждого из двух переданных на вход функции множеств.

Реализация с помощью рекурсии:

Листинг 2: Функция проверки эквивалентности двух множеств

```
(defun check-sets-equal (set1 set2)
  (and
    (is-subset set1 set2)
    (is-subset set2 set1)
  )
)
```

set1 и **set2** - списки-множества.

Листинг 3: Функция проверки вхождения подмножества во множество

```
(defun is-subset (set subset)
  (cond
    (
      (null subset)
    )
    (
      (and
        (contains set (car subset))
        (is-subset set (cdr subset))
      )
    )
  )
)
```

set и **subset** - первое множество и второе множество, для которого проверяется является ли оно подмножеством первого множества.

Листинг 4: Функция проверки вхождения элемента в список

```
(defun contains (lst element)
  (cond
    (
      (null lst)
      nil
    )
    (
      (equal (car lst) element)
    )
    (
      (contains (cdr lst) element)
    )
  )
)
```

lst - список, **element** - элемент, для которого проверяется, входит ли он во список-аргумент.

Рекурсивная реализация является более эффективной по памяти, так как в процессе вычислений не выделяет никаких списочных ячеек. Также она является более эффективной по времени, так как прерывает вычисления, как только обнаруживается, что элемент одного множества не входит в другое.

Примеры работы:

```
> (check-sets-equal '(1 3 2) '(3 2 1))
T
> (check-sets-equal '(1 2) '(3 2 1))
NIL
> (check-sets-equal '(1) '(1))
T
> (check-sets-equal '(E H H H) '(H E))
T
```

Задание 3

Напишите необходимые функции, которые обрабатывают таблицу из точечных пар: (страна . столица), и возвращают по стране - столицу, а по столице - страну.

Реализация с помощью функционалов:

Листинг 5: Функция поиска в списке точечных пар

```
(defun find_in_pairs (lst name)
  (reduce #'(lambda (x y)(or x y))
    (mapcar #'(lambda (x)
        (cond
```

```
5             (
6               (equal (car x) name)
7               (cdr x)
8             )
9             (
10              (equal (cdr x) name)
11              (car x)
12            )
13          )
14        ) lst
15      )
16    )
17  )
```

lst - список точечных пар, **element** - элемент, для которого проверяется, входит ли он в одну из точечных пар списка-аргумента.

С помощью mapcar осуществляется перебор всех точечных пар, возвращается список, состоящий из nil-ов и искомого элемента. Затем с помощью reduce возвращается искомый элемент.

Реализация с помощью рекурсии:

Листинг 6: Функция поиска в списке точечных пар

```
1  (defun find_in_pairs (lst name)
2    (cond
3      (
4        (null lst)
5        nil
6      )
7      (
8        (equal (caar lst) name)
9        (cdar lst)
10     )
11     (
12       (equal (cdar lst) name)
13       (caar lst)
14     )
15     (
16       (find_in_pairs (cdr lst) name)
17     )
18   )
19 )
```

lst - список точечных пар, **element** - элемент, для которого проверяется, входит ли он в одну из точечных пар списка-аргумента.

Рекурсивная реализация является более эффективной по времени, так как если искомый элемент найден, то он сразу возвращается.

Примеры работы:

```
> (find_in_pairs '((moscow . russia) (london . england) (washington . usa))
    'moscow)
RUSSIA
> (find_in_pairs '((moscow . russia) (london . england) (washington . usa))
    'russia)
MOSCOW
> (find_in_pairs '((moscow . russia) (london . england) (washington . usa))
    'kiev)
NIL
```

Задание 7

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

а) все элементы списка - числа,

б) элементы списка - любые объекты.

Реализация с помощью функционалов:

Листинг 7: Функция умножения для списка из чисел

```
(defun mult_num (lst num)
  (mapcar
    #'(lambda (el) (* el num))
    lst
  )
)
```

lst - список чисел, **num** - число, на которое умножаются все числа в списке-аргументе.

Реализация с помощью рекурсии:

Листинг 8: Функция умножения для списка из чисел

```
(defun mult_num (lst num)
  (cond
    (
      (null lst)
      nil
    )
    (
      (cons
        (* (car lst) num)
        (mult_num (cdr lst) num)
      )
    )
```

```
13    )
14 )
```

lst - список чисел, **num** - число, на которое умножаются все числа в списке-аргументе.

В обоих реализациях осуществляется проход по всему списку, умножение каждого элемента на число и создание списочной ячейки под результат умножения.

Реализация с помощью функционалов и рекурсии:

Листинг 9: Функция умножения для списка из любых объектов

```
1  (defun mult_all (lst k)
2    (mapcar #'(lambda (el)
3        (cond
4          (
5            (numberp el)
6            (* el k)
7          )
8          (
9            (atom el)
10           el
11         )
12         (
13           (mult_all el k)
14         )
15       )
16     ) lst
17   )
18 )
```

lst - список, **k** - число, на которое умножаются все числа в списке-аргументе.

Реализация с помощью рекурсии:

Листинг 10: Функция умножения для списка из любых объектов

```
1  (defun mult_all (lst k)
2    (cond
3      (
4        (null lst)
5        nil
6      )
7      (
8        (numberp lst)
9        (* lst k)
10     )
11     (
12       (atom lst)
13       lst
14     )
```

```
15      (
16        (cons
17          (mult_all (car lst) k)
18          (mult_all (cdr lst) k)
19        )
20      )
21    )
22  )
```

lst - список, **k** - число, на которое умножаются все числа в списке-аргументе.

В обоих реализациях осуществляется проход по всему списку, проверка типа каждого элемента:

- если он является числом, то производится умножение на число-аргумент функции и выделение списочной ячейки под результат умножения;
- если он является атомом, то под него выделяется списочная ячейка;
- если он является списком, то к нему рекурсивно применяется текущая функция.

Реализации являются одинаково эффективными, так как и реализация с помощью функционалов обходит список на всех уровнях и выделяет под каждый элемент списочную ячейку, и реализация с помощью рекурсии.

Примеры работы:

```
1  > (mult_num nil 1)
2  NIL
3  > (mult_num '(1) 3)
4  (3)
5  > (mult_num '(1 2 3 4 5) 3)
6  (3 6 9 12 15)
7  > (mult_all nil 2)
8  NIL
9  > (mult_all '(1) 2)
10 (2)
11 > (mult_all '(1 2 3) 3)
12 (3 6 9)
13 > (mult_all '((1 2) 3 (4 (5))) 3)
14 ((3 6) 9 (12 (15)))
15 > (mult_all '(1 2 (a)) 3)
16 (3 6 (A))
```

Лабораторная работа 6

Задание 2

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

Реализация с помощью функционалов и рекурсии:

Листинг 11: Функция уменьшения всех чисел смиска на 10

```
(defun dec_all (lst)
  (mapcar #'(lambda (el)
    (cond
      (
        (numberp el)
        (- el 10)
      )
      (
        (atom el)
        el
      )
      (
        (dec_all el)
      )
      )
    ) lst
  )
)
```

lst - список.

Реализация с помощью рекурсии:

Листинг 12: Функция уменьшения всех чисел смиска на 10

```
(defun dec_all (lst)
  (cond
    (
      (null lst)
      nil
    )
    (
      (numberp lst)
      (- lst 10)
    )
    (
      (atom lst)
      lst
    )
    (
      (cons
        (dec_all (car lst))
        (dec_all (cdr lst))
      )
    )
  )
)
```

lst - список.

В обоих реализациях осуществляется проход по всему списку, проверка типа каждого элемента:

- если он является числом, то из него вычитается 10 и выделяется списочная ячейка;
- если он является атомом, то под него выделяется списочная ячейка;
- если он является списком, то к нему рекурсивно применяется текущая функция.

Реализации являются одинаково эффективными, так как и реализация с помощью функционалов обходит список на всех уровнях и выделяет под каждый элемент списочную ячейку, и реализация с помощью рекурсии.

Примеры работы:

```
> (dec_all nil)
NIL
> (dec_all '(1 2 3 4 5))
(-9 -8 -7 -6 -5)
> (dec_all '(1 (2 3 (4)) (5)))
(-9 (-8 -7 (-6)) (-5))
> (dec_all '(1 (s t (4)) nil  (5)))
(-9 (S T (-6)) NIL (-5))
```

Задание 3

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

Реализация с помощью функционалов:

Листинг 13: Функция поиска первого непустого списка

```
(defun get_first_list (lst)
  (car
    (mapcan #'(lambda (x)
        (cond
          (
            (and
              (listp x)
              (not (null x))
            )
            (cons x nil)
          )
        )
      ) lst
    )
```

```
    )
)
```

lst - список.

С помощью mapcan осуществляется перебор всех элементов списка и формируется список из непустых списков исходного списка, а затем с помощью car возвращается первый из них.

Реализация с помощью рекурсии:

Листинг 14: Функция поиска первого непустого списка

```
(defun get_first_list (lst)
  (cond
    (
      (and
        (listp (car lst))
        (not (null (car lst)))
      )
      (car lst)
    )
    (
      (get_first_list (cdr lst))
    )
  )
)
```

lst - список.

Реализация с помощью рекурсии эффективнее и по памяти, и по скорости, так как не создает вспомогательных списков и возвращает непустой список сразу же, как он встретится.

Примеры работы:

```
> (get_first_list '(1 2 (3) (4 5 (6))))
(3)
> (get_first_list '(((1 2) 3) 4 5))
((1 2) 3)
> (get_first_list '(nil 2 (nil (5)) 6))
(NIL (5))
```

Задание 4

Написать функцию, которая выбирает из заданного списка только числа между двумя заданными границами.

Реализация с помощью функционалов и рекурсии:

Листинг 15: Функция выбора чисел между указанными границами

```
(defun select_between (lst a b)
  (mapcan #'(lambda (x)
      (cond
        (
          (and
            (numberp x)
            (or
              (and (>= x a) (<= x b))
              (and (<= x a) (>= x b))
            )
          )
          (cons x nil)
        )
        (
          (listp x)
          (select_between x a b)
        )
      )
    ) lst
  )
)
```

lst - список, **a**, **b** - числа-границы, между которыми должны быть расположены искомые числа списка-аргумента.

Реализация с помощью рекурсии:

Листинг 16: Функция выбора чисел между указанными границами

```
(defun select_between (lst a b)
  (cond
    (
      (null lst)
      nil
    )
    (
      (and
        (numberp lst)
        (or
          (and (>= lst a) (<= lst b))
          (and (<= lst a) (>= lst b))
        )
      )
      (cons lst nil)
    )
    (
      (listp lst)
      (nconc
        (select_between (car lst) a b)
```

```
21          (select_between (cdr lst) a b)
22        )
23      )
24    )
25  )
```

lst - список, **a**, **b** - числа-границы, между которыми должны быть расположены искомые числа списка-аргумента.

В обоих реализациях осуществляется обход всего списка, если встреченный элемент число и он находится между заданными границами, то под него выделяется списочная ячейка, если элемент список, то к нему рекурсивно применяется текущая функция.

Реализации являются одинаково эффективными по памяти и по времени, так как в обоих осуществляется обход списка на всех уровнях и выделение списочных ячеек под необходимые элементы.

Примеры работы:

```
> (select_between '(1 2 3 4 5 6 7) 3 5)
(3 4 5)
> (select_between '(1 2 (3) (6 (7 8) 2) 9 5) 2 6)
(2 3 6 2 5)
> (select_between '(1 2 (3) (6 (7 8) 2) 9 5) 6 2)
(2 3 6 2 5)
> (select_between nil 1 2)
NIL
```

Задание 5

Написать функцию, вычисляющую декартово произведение двух своих списков-аргументов.

Реализация с помощью функционалов:

Листинг 17: Функция вычисления декартова произведения

```
(defun decart (set1 set2)
  (mapcan #'(lambda (x)
      (mapcar #'(lambda (y)
        (cons x (cons y nil))
        ) set2
      )
    ) set1
  )
)
```

set1 и **set2** - списки-множества.

С помощью mapcar получается декартово произведение одного элемента первого множества на второе множество, с помощью mapcan получается декартово произведение всех элементов первого множества на второе.

Реализация с помощью рекурсии:

Листинг 18: Функция вычисления декартова произведения

```
(defun decart (set1 set2)
  (cond
    (
      (null set1)
      nil
    )
    (
      (nconc
        (decart_element set2 (car set1))
        (decart (cdr set1) set2)
      )
    )
  )
)
```

set1 и **set2** - списки-множества.

Листинг 19: Функция декартова произведения одного элемента на множество

```
(defun decart_element (set el)
  (cond
    (
      (null set)
      nil
    )
    (
      (cons
        (cons el (cons (car set) nil))
        (decart_element (cdr set) el)
      )
    )
  )
)
```

set - список-множество, **el** - элемент, для которого ищется декартово произведение со списком-аргументом.

Реализации одинаково эффективны по памяти и по скорости, так как в обоих реализациях для каждого элемента первого множества вычисляется его декартово произведение на второе множество, а затем все полученные множества объединяются в одно.

Примеры работы:

```
> (decart '(1 2) '(a b c))
((1 A) (1 B) (1 C) (2 A) (2 B) (2 C))
> (decart '(1) '(A))
((1 A))
> (decart '(1) nil)
NIL
```

Задание 6

Почему так реализовано reduce, в чем причина?

```
(reduce #'+ ()) -> 0
```

Причина в том, что reduce должен корректно обрабатывать поданный ему список любого размера, поэтому когда список пуст и аргумент :initial-value не задан, возвращается значение функции-аргумента, которую вызвали без аргументов, а если аргумент :initial-value задан, то возвращается его значение.

Теоретические вопросы

Способы организации повторных вычислений в Lisp

Для организации многократных вычислений в Lisp могут быть использованы функционалы - функции, которые в качестве своего аргумента принимают функциональный объект — функцию, имеющую имя (глобально определенную функцию), или функцию, не имеющую имени (локально определенную функцию).

Также для организации многократных вычислений в Lisp может быть использована рекурсия. Рекурсия — это ссылка на определяемый объект во время его определения.

Различные способы использования функционалов,

При использовании функционального объекта должно быть использовано замыкание контекста функции, которым обеспечивается связывание свободных переменных со значениями. В Lisp используются

- применяющие функционалы (apply, funcall);
- отображающие функционалы (mapcar, maplist, mapcan, mapcon);
- функционалы, являющиеся предикатами;
- функционалы, использующие предикаты в качестве функционального объекта (remove-if, delete-if, remove-if-not, delete-if-not).

Что такое рекурсия? Способы организации рекурсивных функций

Рекурсия — это ссылка на определяемый объект во время его определения. Способы организации рекурсивных функций:

- Хвостовая рекурсия

  Результат формируется не на выходе из рекурсии, а на входе в рекурсию, все действия выполняются до ухода на следующий шаг рекурсии.

```
(defun fun (x)
(cond (end_test1  end_value1)
...
(end_testN  end_valueN)
( (fun reduced_x) )
) )
```

- Рекурсия по нескольким параметрам

```
(defun fun (n x)
(cond  (end_test end_value)
( t  (fun (reduced_n) (reduced_x))
) )
```

- Дополняемая рекурсия

  При обращении к рекурсивной функции используется дополнительная функция не в аргументе вызова, а вне его.

```
(defun fun (x)
(cond (test end_value)
(t (add_fun add_value (fun reduced_x)) )
))
```

- Множественная рекурсия

  На одной ветке происходит сразу несколько рекурсивных вызовов. Количество условий выхода также может зависеть от задачи.

```
(defun fun (x)
(cond  (test end_val)
( t (combine (fun changed1_x)
(fun changed2_x))
)
))
```

Способы повышения эффективности реализации рекурсии

При изучении рекурсии рекомендуется организовывать и отлаживать реализацию отдельных подзадач исходной задачи, обращая внимание на эффективность реализации и качество работы, а потом, при необходимости, встраивать эти функции в более крупные, возможно в виде лямбда-выражений.

Для повышения эффективности рекурсии необходимо правильно организовавывать условия выхода из нее. Основное правило: при построении условного выражения первое условие - это всегда выход из рекурсии, но если условий выхода несколько, то надо думать о порядке их следования. Некачественный выход из рекурсии может привести к переполнению памяти из-за "лишних"рекурсивных вызовов. Кроме того возможна потеря аргумента - кажется что функция возвращает результат и он используется, но на деле результат теряется и ответ неверен.

В целях повышения эффективности рекурсивных функций рекомендуется формировать результат не на выходе из рекурсии, а на входе в рекурсию, все действия выполняя до ухода на следующий шаг рекурсии.