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

Тема Использование функционалов и рекурсии

Студент Сушина А.Д.

Группа ИУ7-61б

Оценка (баллы) _____

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.

Ход работы

Практическое задание: задания: №2 , №3, (доп)№7 из лабораторной 5;

задания: №2, №3, №4, №5, №6 из лабораторной 6

Лаб. работа № 5

2. Написать предикат set-equal, который возвращает t, если два его множество-

аргумента содержат одни и те же элементы, порядок которых не имеет значения.

```
(defun set-equal(a b) (and (subsetp a b) (subsetp b a)))
```

> (set-equal `(1 2 3 4) `(4 3 2 1)) → T

> (set-equal `(1 2 3 4) `(4 3 2)) → Nil

Функция set-equal получает на вход два агрумента — два списка a и b. Проверяет содержат они одни и те же элементы, порядок которых не имеет значения. Для реализации используется функция subsetp стандартной библиотеки, которая проверяет, является ли одно множество подмножеством другого. Проверяет выполнение двух условий одновременно: a является подмножеством b и b является подмножеством a. Если оба условия выполняются, то оба списка содержат одни и те же элементы.

Рекурсивная реализация

```
(defun is_in(a list) (
                cond ((null list) Nil)
                ((equal a (car list)) T)
                (t (is_in a (cdr list))) ))
(defun my_subsetp(a b) (
                  cond ((null (cdr a)) (is_in (car a)))
                  ((not(is_in (car a) b)) nil)
                  (t (my_subsetp (cdr a) b))))
(defun set-equal_rec(a b) (and (my_subsetp a b) (my_subsetp b a)))
```

> (set-equal_rec `(1 2 3 4) `(4 3 2 1)) → T

> (set-equal_rec `(1 2 3 4) `(4 3 2)) → Nil

Функция is_in принимает на вход два аргумента — элемент и целевой список. Функция проверяет содержится ли элемент в целевом списке. Если целевой список пуст, то он не может содержать элемент, соответственно функция возвращает Nil. Если список не пуст, то

сравним элемент с головой списка. Если они равны, то функция возвращает T. В иных случаях производится рекурсивный вызов функции is_in с двумя аргументами: элемент и хвост списка. Все проверки реализованы с помощью cond.

Функция my_subsetp принимает на вход два аргумента — два списка a и b. Проверяет содержит ли список b все элементы списка a. Сначала проверим, является ли хвост списка a списком. Если нет, проверим содержится ли голова списка a в списке b. Если содержится, то вернем T(список a содержится в b), иначе Nil (не содержится). Если список не пустой, то проверим, содержится ли голова списка a в списке b. Если не содержит, возвращаем Nil(не содержит хотя бы один элемент — a не является подмножеством). Иначе рекурсивно вызываем функцию my_subsetp аргументами : головой списка a и списком b.

Функция set-equal_req принимает на вход два аргумента a b — два списка. Проверяет содержат они одни и те же элементы, порядок которых не имеет значения. Проверяет выполнение двух условий одновременно: a является подмножеством b и b является подмножеством a. Если оба условия выполняются, то оба списка содержат одни и те же элементы.

Функционалы

```
(defun my_sub_2(a b)
   (reduce #'(lambda(a x) (and a x))
         ( mapcar #'(lambda(x)
               (reduce #'(lambda(a y) (or a y))
                     (mapcar #'(lambda(z) (equal z x)) b)))
          a)
   ))
(defun set-equal_func(a b) (and ( my_sub_2 a b) ( my_sub_2 b a)))
```

> (set-equal_func `(1 2 3 4) `(4 3 2 1)) → T

> (set-equal_func `(1 2 3 4) `(4 3 2)) → Nil

Функция my_sub_2 принимает два аргумента — два списка. Проверяет содержатся ли все элементы списка a в списке b. Для каждого элемента X списка A вычисляется значение T либо Nil, которое формируется следующим образом. Для каждого элемента списка B проверяется равен ли он X. Из полученных значений с помощью функции reduce формируется значение T(если хоть один элемент равен, т. е. X содержится в B) или Nil(иначе). Из полученных значений формируется значение T (если для всех X получено значение T, а соответственно все элементы входят в B) или Nil (иначе)

Функция set-equl_func принимает на вход 2 элемента — 2 списка. Проверяет содержат они одни и те же элементы, порядок которых не имеет значения. Проверяет выполнение двух условий одновременно: a является подмножеством b и b является подмножеством a. Если оба условия выполняются, то оба списка содержат одни и те же элементы.

3. Напишите необходимые функции, которые обрабатывают таблицу из точечных пар:

(страна. столица), и возвращают по стране - столицу, а по столице - страну.

Рекурсия

```
(defun find_by(c list)
   (cond ((null list) Nil)
      ((equal c (caar list)) (cdar list))
      ((equal c (cdar list)) (caar list))
      (t (find_by c (cdr list))) ))
```

> (find_by `moscow `((russia . moscow) (italy . rim)) → russia

> (find_by `russia `((russia . moscow) (italy . rim)) → moscow

Функция find_by принимает на вход два аргумента: элемент и список. Производит поиск страны по столицы и столицы по стране. Если список пустой, значит элемент не найден, возвращается Nil. Проверяем совпадает ли голова головы списка с искомым элементом, если да, возвращаем хвост головы списка. Проверяем совпадает ли хвост головы списка с элементом, если да, то возвращаем голову головы списка. Иначе продолжаем поиск с помощью рекурсивного вызова функции find_by с аргументами элемент и хвост списка.

Функционалы

```
(defun find_by_func(val lst)
   (find-if (lambda (x) (not (null x)))
         (mapcar (lambda (pair) (cond ((equal (car pair) val) (cdr pair))
                                      ((equal (cdr pair) val) (car pair))))
            lst)))
```

> (find_by_func `moscow `((russia . moscow) (italy . rim)) → russia

> (find_by_func `russia `((russia . moscow) (italy . rim)) → moscow

Функция find_by_func принимает на вход два аргумента: элемент и список. Производит поиск страны по столицы и столицы по стране. В ходе вычисления функция mapcar формирует список, в котором хранится Nil или результат, полученный по ключу. Функция find_by возвращает элемент не равный Nil, то есть значение.

7. Напишите функцию, которая умножает на заданное число-аргумент все числа

из заданного списка-аргумента, когда

а) все элементы списка --- числа,

Функционалы

```
(defun mulall (mul lst)
        (mapcar (lambda (x) (* x mul)) lst)
)
```

> (mulall `4 `(1 2 3 4)) → (4 8 12 16)

Функция mulall получает на вход два параметра — множитель и список. С помощью функции mapcar выполняется умножение каждого элемента списка на множитель.

Рекурсия

```
(defun mulall_rec (mul lst)
   ( cond  ((null lst) nil)
      ( t (cons (* mul (car lst)) (mulall_rec mul (cdr lst))))
    )
)
```

> (mulall_rec `4 `(1 2 3 4)) → (4 8 12 16)

Функция mulall_rec получает на вход два параметра — множитель и список. Если список пустой, то возвращает Nil. Иначе объединяет результат умножения и рекурсивный вызов функции mulall_rec с параметрами множитель и хвост списка.

Рекурсия (2 вариант)

```
(defun mulall_rec_r2 (mul lst rst)
   ( cond ((null lst)  (cdr rst))
        (t (mulall_rec_r2 mul (cdr lst) (append rst  (cons (* mul (car lst)) nil)))))
)
(defun mulall_r2 (mul lst)
   (mulall_rec_2 mul lst (cons nil nil)))
```

Функция mulall_rec_r2 получает на вход три параметра — множитель и список и доп параметр для результата. Если список пустой, то возвращает результат. Иначе выполняет рекурсивный вызов функции mulall_rec_r2 с параметрами множитель, хвост списка и результат, к которому в конец добавлен результат умножения.

Функция mulall_r2 получает на вход два аргумента — множитель и список. Вызывает функцию mulall_rec_2 с начальным значением результата (nil)

6) элементы списка -- любые объекты.

Функционалы

```
(defun mulall_2 (mul lst)
        (mapcar #'(lambda (x)
                (cond
```

```
                ((numberp x) (* x mul))
                ((listp x) (mulall_2 mul x))
                (t x)))
                    lst
        )
)
```

> (mulall_2 2 `(1 (2 (3)) 4)) → (2 (4 (6)) 8)

Функция mulall_2 получает на вход два параметра — множитель и список. Затем к каждому элементу списка применяется функция, которая проверяет, является ли элемент числом или списком. Если выполнено первое, то производит умножение. Если второе, то производит рекурсивный вызов mulall_2 с параметрами множитель и элемент. Иначе возвращает элемент.

```
(defun mulall_rec_2 (mul lst)
   ( cond  ((null lst) nil)
      ((listp (car lst)) (cons (mulall_rec_2 mul (car lst))  (mulall_rec_2 mul (cdr lst))))
      ((numberp (car lst)) (cons (* mul (car lst)) (mulall_rec_2 mul (cdr lst))))
      (t (cons (car lst) (mulall_rec_2 mul (cdr lst))))
    )
)
```

> (mulall_rec_2 2 `(1 (2 (3)) 4)) → (2 (4 (6)) 8)

Функция mulall_rec_2 получает на вход два параметра — множитель и список. Если список пустой, возвращается Nil. Если элемент является списком, то объединяются результаты рекурсивных вызовов mulall_rec_2 с параметрами множитель и голова списка и множитель и хвост списка. Если элемент является числом, то объединяются результат умножения головы на множитель и результат рекурсивного вызова функции mulall_rec_2 с параметрами множитель и хвост списка.

Лаб. работа № 6

2. Напишите функцию, которая уменьшает на 10 все числа из списка

аргумента этой функции.

Рекурсия

```
(defun minus10(lst)
   ( cond  ((null lst) nil)
      ((numberp (car lst)) (cons (- (car lst) 10) (minus10 (cdr lst))))
      ((listp (car lst)) (cons (minus10 (car lst)) (minus10 (cdr lst))))
```

```
      (t (cons (car lst) (minus10 (cdr lst))))
   )
)
```

> (minus10 `(10 (10 (11)) 12)) → (0 (0 (1)) 2)

Функция minus10 на вход получает список. Если список пустой, возвращает Nil. Если голова списка является числом, то результат вычисления присоединяется к рекурсивному вызову для хвоста списка. Если это список, объединяем результаты рекурсивного вызова функции для головы и рекурсивного вызова функции для хвоста списка. Иначе присоединяет голову к рекурсивному вызову от хвоста списка.

Функционалы

```
(defun minus10_f(lst)
   ( mapcar #'(lambda (x)
                  (cond ((numberp x) (- x 10))
                     ((listp x) (minus10 x))
                    (t x)))
         lst
    )
)
```

> (minus10_f `(10 (10 (11)) 12)) → (0 (0 (1)) 2)

Функция minus10_f принимает на вход один аргумент — список. С помощью функции mapcar к каждому элементу применяется функция, которая проверяет является ли элемент числом или списком. Если элемент является числом, то вычитает из него 10. Если списком, вызывает функцию рекурсивно для элемента. Иначе возвращает сам элемент.

3. Написать функцию, которая возвращает первый аргумент списка -аргумента.

который сам является непустым списком.

Рекурсия

```
(defun get_first_element (lst)
        (cond ( (null lst) nil)
                ((and   (listp (car lst)) (not (null (car lst)))) (car lst))
                 (t (get_first_element (cdr lst)))
        )
)
```

> (get_first_element `(1 () 3 (2) 5)) → (2)

Функция get_first_element принимает на вход список. Если список пустой, возвращается Nil. Если голова списка является непустым списком, возвращается эта голова. Иначе рекурсивно вызывается функция от хвоста списка.

Функционалы

```
(defun get_first (lst)
   ( find-if (lambda (x) (and (listp x) (not (null x)))) lst
   )
)
```

> (get_firs `(1 () 3 (2) 5)) → (2)

Функция get_first принимает на вход список. Для каждого элемента выполняется проверка является ли он непустым списком. Возвращается первый элемент, удовлетворяющий условию.

4. Написать функцию, которая выбирает из заданного списка только те числа,

которые больше 1 и меньше 10.

(Вариант: между двумя заданными границами.)

Рекурсия

```
(defun getfromto (lst from to)
   ( cond ((null lst) Nil)
        ((listp (car lst)) (nconc (getfromto (car lst) from to)
                                  (getfromto (cdr lst) from to)))
        ((and (numberp (car lst))
              (< from (car lst))
              (> to (car lst)))
         (cons (car lst) (getfromto (cdr lst) from to)))
        (t (getfromto (cdr lst) from to))
   )
)
```

> (getfromto `(1 2 (3 4 5) 11 21) 1 10) → (2 3 4 5)

Функция getfromto принимает на вход три аргумента: список, начальное значение и конечное значение. Составляет список из всех элементов, входящих в промежуток. Сначала проверяет пустой ли список, если пустой, то возвращает Nil. Если голова списка является списком, то объединяет результаты рекурсивного вызова функции для головы списка и рекурсивного вызова функции для хвоста списка. Если Голова списка является числом и входит в

промежуток, то голова объединяется с результатом рекурсивного вызова функции для хвоста списка. Иначе производится рекурсивный вызов функции.

Функционалы

```
(defun select_between_inner (lst left right result)
        (mapcar         #'(lambda (x)
                                        (cond   ((listp x) (select_between_inner x left right result))
                                                ((and   (numberp x) (> x left) (< x right))
                                                        (nconc result (cons x nil))
                                                )
                                        )
                                )
                        lst
        )
        (cdr result)
)
(defun getfromto_2 (lst left right);
        (select_between_inner lst left right (cons nil nil))
)
```

> (getfromto_2 `(1 2 (3 4 5) 11 21) 1 10) → (2 3 4 5)

Функция select_between_inner принимает на вход четыре аргумента: список, левую и правую границы промежутка и результирующий список. С помощью функции mapcar к каждому элементу списка применяется функция, проверяющая является ли элемент списком или числом. Если элемент - список, то выполняет рекурсивный вызов функции. Если число и входит в промежуток, то присоединяет его к результату.

Функция getfromto_2 принимает на вход три аргумента: список, левую и правую границы промежутка. Выполняет вызов функции select_between_inner с аргументами список, границы промежутка и список из Nil

5. Написать функцию, вычисляющую декартово произведение двух своих списков-аргументов. (Напомним, что A x B это множество всевозможных пар (a b), где a принадлежит A, принадлежит B.)

Функционалы

```
(defun decart (a b)
        (mapcan #' (lambda (x) (mapcar #' (lambda (y) (list x y)) b ) ) a )
)
```

> (decart `(a b) `(1 2)) → ((a 1) (a 2) (b 1) (b 2))

Функция decart принимает два аргумента - два списка. С помощью mapcar для каждого элемента из первого списка собираются все возможные пары с элементами второго списка. Mapcan объединяет результаты для всех элементов первого списка в один результирующий.

Рекурсия

```
(defun dec_rec(el lst)
   ( cond ((null lst) Nil)
      (t (cons (cons el (cons (car lst) nil)) (dec_rec el (cdr lst))))
   )
)


(defun decart_rec(a b)
   ( cond ( (null a) nil)
      (t (nconc (dec_rec (car a) b) (decart_rec (cdr a) b)))
   )
)
```

> (decart_rec `(a b) `(1 2)) → ((a 1) (a 2) (b 1) (b 2))

Функуия dec_rec принимает на вход два аргумента - элемент и список. Собирает все пары для этого элемента с элементами списка. Если список пуст, возвращает nil. Если не пуст, то создает список из двух элементов элемент и голова списка и объединяет его с результатом рекурсивного вызова функции для хвоста списка.

Функция decart_rec принимает на вход два аргумента — два списка. Если первый список пуст, возвращает nil. Иначе Объединяет результат выполнения функции dec_rec с результатом рекурсивного вызова функции для хвоста первого списка и второго списка.

6. Почему так реализовано reduce, в чем причина?

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

Сначала функция проверяет список-аргумент. Если он пуст, возвращается значение функции при отсутствии аргументов. Также reduce использует аргумент :initial-value. Этот аргумент определяет значение, к которому будет применена функция при обработке первого элемента списка-аргумента. Если список-аргумент пуст, то будет возвращено значение initial-value.

Результатом вычисления функции + без аргументов будет 0, а результатом вычисления функции * без аргументов будет 1, т.к. это нейтральные элементы для данных операций

Теоретические вопросы:

· Способы организации повторных вычислений в Lisp

Для организации многократных вычислений в Lisp могут быть использованы функционалы — функции, которые особым образом обрабатывают свои аргументы. 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

Функционалы используются в Lisp для организации повторных вычислений. Также могут использоваться для обработки списковых таблиц с помощью функционалов.

· **Что такое рекурсия? Способы организации рекурсивных функций**

Рекурсия — это ссылка на определяемый объект во время его определения.

В LISP существует классификация рекурсивных функций:

- простая рекурсия - один рекурсивный вызов в теле
- рекурсия первого порядка - рекурсивный вызов встречается несколько раз
- взаимная рекурсия - используется несколько функций, рекурсивно вызывающих друг друга.

Виды рекурсии:

- Хвостовая. Результат формируется не на выходе из рекурсии, а на входе в рекурсию, все действия выполняются до ухода на следующий шаг рекурсии.
- Дополняемая. При обращении к рекурсивной функции используется дополнительная функция не в аргументе вызова , а вне его.
- Множественная. На одной ветке происходит сразу несколько рекурсивных вызовов. Количество условий выхода также может зависеть от задачи.
- Взаимная.
- рекурсии высокого порядка.

При организации рекурсии можно использовать как функции с именем, так и локально определенные с помощью лямбда выражений функции. Кроме этого, при организации рекурсии можно использовать функционалы или использовать рекурсивную функцию внутри функционала.

· Способы повышения эффективности реализации рекурсии

Один из методов повышения эффективности рекурсии является организация хвостовой рекурсии. Для этого может потребоваться использовать дополнительные параметры. Такая рекурсия может быть путём формальной и гарантированно корректной перестройки кода заменена на итерацию. Такая оптимизация реализована во многих оптимизирующих компиляторах, а в трансляторах Scheme, одного из диалектов Lisp, такая оптимизация является обязательной.