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»

Дисциплина: «Функциональное и логическое программирование»

Лабораторная работа №10

Студент: Левушкин И. К.
Группа: ИУ7-62Б
Преподаватели: Толпинская Н. Б.,
Строганов Ю. В.

Москва, 2020 г.

6.7. Пусть list-of-list список, состоящий из списков. Написать функцию, которая вычисляет сумму длин всех элементов list-of-list, т.е. например для аргумента ((1 2) (3 4)) -> 4.

Реализация задания

```
(defun add-list-deep (lst depth)
    (cond ((null lst) depth)
          (t (add-list-deep (cdr lst) (+ 1 depth)))
    )
)

(defun summ-length-help (list-of-list summ)
    (cond ((null list-of-list) summ)
        (t (summ-length-help (cdr list-of-list) (+ (add-list-deep (car list-of-list) 0) summ)))
    )
)

(defun summ-length (list-of-list)
    (summ-length-help list-of-list 0)
)
```

Рис. 1: Рекурсивная реализация функции, вычисляющей сумму длин всех элементов списка

Назначение параметров функций

- Функция summ-length - функция-обретка, запускающая рекурсивную функцию summ-length-help с начальным параметром суммы summ = 0
- Функция summ-length-help проверет, пустой ли список, если пустой, то возвращает summ, если нет, то подсчитывает длинну (глубину) очередного аргумента списка, вызывая при этом рекурсивную функцию add-list-deep с начальным параметром deep = 0
- Функция add-list-deep считает глубину списка lst (до тех пор, пока lst не nil)

Результаты работы

Выражение	Результат
'((1 2) (3 4))	4
'((1 2) NIL (3 4))	4
'((1 2) NIL (3 4 (5 6)))	5
'()	0
'((3 4 5 6 7) () (1 2))	7

6.8. Написать рекурсивную версию (с именем reg-add) вычисления суммы чисел заданного списка. Например: (reg-add (2 4 6)) -> 12

Реализация задания

Так как в условии задачи не сказано, является ли список-аргумент списком чисел, будем считать, что элементы списка - любые объекты.

```
(defun reg-add-help (lst summ)
    (cond ((null lst) summ)
          (t  (reg-add-help (cdr lst) (if (listp (car lst))
                                          (+ (reg-add (car lst)) summ) (+ summ (car lst)))
              )
          )
    )
)

(defun reg-add (lst)
    (reg-add-help lst 0)
)
```

Рис. 2: Рекурсивная реализация функции reg-add

Назначение параметров функций

- Функция reg-add - функция-обертка, запускающая рекурсивную функцию reg-add-help с начальным параметром summ = 0
- Функция reg-add-help - функция, подсчитывающая сумму чисел заданного списка. Если очередной элемент списка является списком, считает сумму чисел уже внутреннего списка, запуская функцию reg-add для этого списка, и добавляет к уже имеющейся

Результаты работы

Выражение	**Результат**
'(1 2 3 4 5 6)	21
'(1 2 3 (5 6 7) 3 2)	29
'(1 2 3 () 3 2)	11
'()	0

6.9. Написать рекурсивную версию с именем recnth функции nth.

Реализация задания

```
(defun recnth-help (lst n count)
    (cond ((null lst) count)
          ((equal n count) (car lst))
          (t (recnth-help (cdr lst) n (+ count 1)))
    )
)

(defun recnth (n lst)
    (cond ((null lst) nil)
          ((>= n (length lst)) nil)
          (t (recnth-help lst n 0))
    )
)
```

Рис. 3: Рекурсивная реализация функции recnth

Назначение параметров функций

- Функция recnth - функция-обертка, проверяющая lst на пустой список и, меньше ли входной параметр n длины списка lst. Если не пуст и меньше, то запускает рекурсивную функцию recnth-help с начальным параметром count = 0 (глубина списка), иначе nil

Результаты работы

Выражение	**Результат**
2 '(1 2 3 4 5)	3
5 '(1 2 3 4 5)	NIL
0 '(1 2 3 4 5)	1
1 '(1)	NIL
1 '()	NIL
3 '(1 2 3 (1 2 3) 4 5)	(1 2 3)

6.10. Написать рекурсивную функцию alloddr, которая возвращает t когда все элементы списка нечетные.

Реализация задания

Так как в условии задачи не сказано, является ли список-аргумент списком чисел, будем считать, что элементы списка - любые объекты.

```
(defun alloddr-help-low (obj)
    (cond ((null obj) nil)
          ((listp obj) (alloddr-help obj))
          (t (if (oddp obj) t nil))
    )
)

(defun alloddr-help (lst)
    (cond ((null lst) t)
          (t  (if (alloddr-help-low (car lst)) (alloddr-help (cdr lst)) nil
              )
          )
    )
)

(defun alloddr (lst)
    (cond ((null lst) nil)
        (t (alloddr-help lst))
    )
)
```

Рис. 4: Рекурсивная реализация функции alloddr

Назначение параметров функций

- Функция alloddr - функция-обертка, проверяющая список на nil. Если не пуст, то запускает рекурсиную функцию alloddr-help, где при проверке списка lst на nil, будет возвращаться t, так как будет считаться, что функция перебрала все элементы списка и не нашла ни одного четного
- Функция alloddr-help - рекурсивная функция, проходящая по всем элементам списка и проверяющая с помощью функции alloddr-help-low, является ли какие-нибудь элементы четными или нет
- Функция alloddr-help-low - функция, проверяющая является ли obj списком. Если да, то запускает функцию alloddr-help для него, если нет, то проверяет его на четность нечетность

Результаты работы

Выражение	**Результат**
'(1 2 3 4 5 6)	NIL
'(1 3 5)	T
'(1)	T
'()	NIL
'(2)	NIL
'(1 3 (5 7))	T
'(1 3 (5 6 7))	NIL

6.11. Написать рекурсивную функцию, относящуюся к хвостовой рекурсии с одним тестом завершения, которая возвращает последний элемент списка - аргументы.

Реализация задания

```
(defun last-elem (lst)
    (cond ((null (cdr lst)) (car lst))
        (t (last-elem (cdr lst)))
    )
)
```

Рис. 5: Рекурсивная реализация функции, возвращающей последний элемент списка - аргументы

Назначение параметров функций

- Функция last-elem - рекурсивная функция с одним варинтом завершения - если следующий элемент списка пустой (что означает, что текущий элемент списка - последний)

Результаты работы

Выражение	**Результат**
'(1 2 3 4 5 6)	6
'(1 2 3 4 (1 2 3))	(1 2 3)
()	NIL

6.12. Написать рекурсивную функцию, относящуюся к дополняемой рекурсии с одним тестом завершения, которая вычисляет сумму всех чисел от 0 до n-ого аргумента функции.

Вариант:

1. от from-аргумента функции до последнего,
2. от from-аргумента функции до to-аргумента с шагом d.

Реализация задания

Так как в условии задачи не сказано, является ли список-аргумент списком чисел, будем считать, что элементы списка - числа, поскольку реализация функций не сильно изменится (добавится лишь еще одна проверка в cond).

Так как задание лабораторной работы состоит в том, чтобы реализовать поставленные задачи, используя хвостовую рекурсию, было решено использовать ее, а не дополняемую рекурсию, как в условии задачи.

```
;0######

;n - не включительно
(defun sum-elem-from-help (lst n pos summ)
    (cond ((null lst) summ)
          ((equal n pos) summ)
          (t (sum-elem-from-help (cdr lst) n (+ pos 1) (+ summ (car lst))))
    )
)

(defun sum-elem-from (lst n)
    (cond ((null lst) nil)
        (t (sum-elem-from-help lst n 0 0))
    )
)
```

Рис. 6: Рекурсивная реализация функции, вычисляющей сумму всех чисел от 0 до n-ого аргумента функции

```
;1######

;используется sum-elem-from-help из предыдущего
(defun sum-elem-to (lst from)
    (cond ((null lst) nil)
          ((>= from (length lst)) 0)
        (t (sum-elem-from-help (reverse lst) (- (length lst) from) 0 0))
    )
)
```

Рис. 7: Рекурсивная реализация функции, вычисляющей сумму всех чисел от from-аргумента до последнего

```
;2######


;to - не включительно
(defun sum-elem-from-to-h-help (lst to h pos summ)
    (cond ((>= pos to) summ)
          (t (sum-elem-from-to-h-help lst to h (+ pos h) (+ summ (
              let ((elem (nth pos lst)));ищем элемент списка на позиции pos
                  (if elem elem 0)
          ))))
    )
)

(defun sum-elem-from-to-h (lst from to h)
    (cond ((null lst) nil)
          ((> from to) 0)
        (t (sum-elem-from-to-h-help lst to h from 0))
    )
)
```

Рис. 8: Рекурсивная реализация функции, вычисляющей сумму всех чисел от from-аргумента до to-аргумента с шагом d

Назначение параметров функций

- Функция sum-elem-from - функция - обертка, проверяющая - пустой ли список lst. Если нет, то запускает рекурсивную функцию sum-elem-from-help с начальными параметрами pos (позиция в списке) = 0, summ (сумма чисел) = 0, иначе nil
- Функция sum-elem-from-help - рекурсивная функция, считающая сумму всех чисел от 0 и до n-ого аргумента функции. Выход из рекурсии осуществляется следующим образом: если список пуст; позиция pos в списке

стала равна входному параметру n

- Функция sum-elem-to - функция, аналогичная по логике функции sum-elem-from. Поэтому она переворачивает список lst и вызывает предыдущую функцию sum-elem-from-help с этим списком и (length(lst) - from) позицией в списке, до какой необходимо считать

- Функция sum-elem-from-to-h - функция-обертка, проверяющая - пустой ли список lst, и больше ли параметр from параметра to. Если не пустой, и не больше, то вызывает рекурсивную функцию sum-elem-from-to-h-help с начальными параметрами pos (текущая позиция в списке) = from и summ = 0, иначе nil, если пустой, и 0, если больше

- Функция sum-elem-from-to-h-help - рекурсивная функция, являющаяся по сути реализацией цикла вида: for (int i = from; i < to; i + h) в C++, ищущая на каждой итерации элемент в списке lst на позиции pos с помощью функции nth.

Результаты работы

Выражение	**Результат sum-elem-from**
'(1 2 3 4 5 6 7 8) 4	10
'(1 2 3 4 5 6 7 8) 7	36
'(1 2 3 4 5 6 7 8) 0	0
'(1 2 3 4 5 6 7 8) 10	36
'() 5	NIL

Выражение	**Результат sum-elem-to**
'(1 2 3 4 5 6 7 8) 4	26
'(1 2 3 4 5 6 7 8) 7	8
'(1 2 3 4 5 6 7 8) 0	36
'(1 2 3 4 5 6 7 8) 10	0
'() 5	NIL

Выражение	**Результат sum-elem-from-to-h**
'(1 2 3 4 5 6 7 8 9 10) 3 9 1	39
'(1 2 3 4 5 6 7 8 9 10) 3 9 2	18
'(1 2 3 4 5 6 7 8 9 10) 3 9 3	11
'(1 2 3 4 5 6 7 8 9 10) 9 3 1	0
'(1 3 5 9) 0 4 2	6
'(1) 0 0 1	0
'() 0 0 0	NIL

6.13. Написать рекурсивную функцию, которая возвращает последнее нечетное число из числового списка, возможно создавая некоторые вспомогательные функции.

Реализация задания

```
;13----------------------------------------

(defun last-oddp-number-help (lst last-number)
    (cond ((null lst) last-number)
          (t (last-oddp-number-help (cdr lst) (if (oddp (car lst)) (car lst) last-number)))
    )
)

(defun last-oddp-number (lst)
    (last-oddp-number-help lst nil)
)
```

Рис. 9: Рекурсивная реализация функции, возвращающей последнее нечетное число из числового списка

Назначение параметров функций

- Функция last-oddp-number - функция - обертка, вызывающая рекурсивную функцию last-oddp-number-help с параметром last-number = nil
- Функция last-oddp-number-help - рекурсивная функция, возвращающая последнее нечетное число из числового списка.
- Если элемент нечетный, то вызываем снова эту функцию с last-number, равным этому элементу

Результаты работы

Выражение	**Результат**
'(1 2 3 4 5 6 7)	7
'(1 2 3 4 5 6)	5
'(1 2 4 6 8 10)	1
'(2 4 6 8 10)	NIL
'(1)	1
'(2)	NIL
'()	NIL

6.14. Используя cons-дополняемую рекурсию с одним тестом завершения, написать функцию которая получает как аргумент список чисел, а возвращает список квадратов этих чисел в том же порядке.

Реализация задания

Так как задание лабораторной работы состоит в том, чтобы реализовать поставленные задачи, используя хвостовую рекурсию, а в условии задачи, используя cons-дополняемую рекурсию, было решено сделать 2 реализации: используя хвостовую и cons-дополняемую рекурсию.

```
(defun square-list-help (lst result)
    (cond ((null lst) result)
        (t (square-list-help (cdr lst) (cons (* (car lst) (car lst)) result)))
    )
)

(defun square-list (lst)
    (reverse (square-list-help lst nil))
)

;cons-дополняемая
(defun cons-square-list (lst)
    (cond ((null lst) nil)
        (t (cons (* (car lst) (car lst)) (cons-square-list (cdr lst))))
    )
)
```

Рис. 10: Рекурсивные реализации функций, возвращающих список квадратов чисел-аргументов в том же порядке

Назначение параметров функций

- Функция square-list - функция-обертка, вызывающая рекурсивную функцию square-list-help с начальным параметром result = nil, и переворачивающая результирующий список функцией reverse
- Функция square-list-help - рекурсивная функция, проходящаяся по всем элементам списка, возводя их в квадрат и добавляя их в голову result

Результаты работы

Выражение	Результат
'(1 2 3 4 5 6)	(1 4 9 16 25 36)
'(1 2 3 0)	(1 4 9 0)
'(4)	(16)
'()	NIL

6.15. Написать функцию с именем select-odd, которая из заданного списка выбирает все нечетные числа. Вариант 1: select-even, вариант 2: вычисляет сумму всех нечетных чисел(sum-all-odd) или сумму всех четных чисел (sum-all-even) из заданного списка.

Реализация задания

Было решено реализовать функции select-odd и sum-all-odd (нечетные числа):

```
;для списка чисел
(defun select-odd-help (lst answer)
    (cond ((null lst) answer)
          (t (select-odd-help (cdr lst) (if (oddp (car lst))
                                            (cons (car lst) answer) answer)))
    )
)

(defun select-odd (lst)
    (reverse (select-odd-help lst nil))
)



(defun sum-all-odd-help (lst answer)
    (cond ((null lst) answer)
          (t (sum-all-odd-help (cdr lst) (if (oddp (car lst))
                                             (+ (car lst) answer) answer)))
    )
)

(defun sum-all-odd (lst)
    (cond ((null lst) nil)
    (t (sum-all-odd-help lst 0))
    )
)
```

Рис. 11: Рекурсивная реализация функций для списка чисел

```
;для объекта


(defun select-odd-help (lst answer)
    (cond ((null lst) answer)
          ((listp (car lst)) (select-odd-help (cdr lst) (select-odd-help (car lst) answer)))
          (t (select-odd-help (cdr lst) (if (oddp (car lst)) (cons (car lst) answer) answer)))
    )
)

(defun select-odd (lst)
    (select-odd-help lst nil)
)
```

Рис. 12: Рекурсивная реализация функции select-odd для списка объектов

```
(defun sum-all-odd-help (lst answer)
    (cond ((null lst) answer)
          ((listp (car lst)) (sum-all-odd-help (cdr lst) (sum-all-odd-help (car lst) answer)))
          (t (sum-all-odd-help (cdr lst) (if (oddp (car lst)) (+ (car lst) answer) answer)))
    )
)

(defun sum-all-odd (lst)
    (cond ((null lst) nil)
    (t (sum-all-odd-help lst 0))
    )
)
```

Рис. 13: Рекурсивная реализация функции sum-all-odd для списка объектов

Назначение параметров функций

- Функция select-odd - функция-обертка, вызывающая рекурсивную функцию select-odd-help с начальным параметром answer = nil, и переворачивающая результирующий список функцией reverse
- Функция select-odd-help - рекурсивная функция, проходящаяся по всем элементам списка, проверяющая их на нечетность и добавляя их в голову answer, если нечетное
- Функция sum-all-odd - функция-обертка, проверяющая пустой ли список и вызывающая рекурсивную функцию sum-all-odd-help с начальным параметром answer = 0, если не пуст, иначе nil
- Функция sum-all-odd-help - рекурсивная функция, проходящая по всем элементам списка, проверяющая их на нечетность и прибавляет их к result, если нечетное

Результаты работы

Функция select-odd:

Выражение	**Результат для списка чисел**	**Результат для списка объектов**
'(1 2 3 4 5 6 7)	(1 3 5 7)	(1 3 5 7)
'(2 4 6)	NIL	NIL
'(1 2 3 4 5 6)	(1 3 5)	(1 3 5)
'(1)	(1)	(1)
'()	NIL	NIL
'(1 2 3 4 5 (3 5) 7)	—	(7 5 3 5 3 1)
'(1 2 3 4 5 (3 (2) 5) 7)	—	(7 5 3 5 3 1)

Функция sum-all-odd:

Выражение	**Результат для списка чисел**	**Результат для списка объектов**
'(1 2 3 4 5 6 7)	16	16
'(2 4 6)	0	0
'(1 2 3 4 5 6)	9	9
'(1)	1	1
'()	NIL	NIL
'(1 2 3 4 5 (3 5) 7)	—	24
'(1 2 3 4 5 (3 (2) 5) 7)	—	24

Ответы на вопросы

Способы организации повторных вычислений в Lisp:

1. Использование функционалов.
2. Использование рекурсиии.

Что такое рекурсия?

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

Классификация рекурсивных функций в Lisp:

1. Простая рекурсия – один рекурсивный вызов в теле.
2. Рекурсия первого порядка – рекурсивный вызов встречается несколько раз.
3. Взаимная рекурсия – используется несколько функций, рекурсивно вызывающих друг друга.

Различные способы организации рекурсивных функций и порядок их реализации.

Способы организации рекурсивных функций:

1. *Хвостовая рекурсия.* 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. Это и есть хвостовая рекурсия.

Листинг 1: Хвостовая рекурсия

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

2. *Рекурсия по нескольким параметрам.*

Листинг 2: Рекурсия по нескольким параметрам

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

3. *Дополняемая рекурсия* – при обращении к рекурсивной функции используется дополнительная функция не в аргументе вызова, а вне его.

Листинг 3: Дополняемая рекурсия

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

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

Листинг 4: Множественная рекурсия

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

Способы повышения эффективности реализации рекурсии.

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. Это и есть хвостовая *рекурсия*.

Для превращения не хвостовой рекурсии в хвостовую и в целях формирования результата (результирующего списка) на входе в рекурсию рекомендуется использовать дополнительные (рабочие) параметры. При этом становится необходимым создать функцию-оболочку для реализации очевидного обращения к функции.