



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

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Лабораторная работа № 9

| | |
|---------------|--|
| Дисциплина | Функциональное и логическое программирование |
| Студент | Сиденко А.Г. |
| Группа | ИУ7-63Б |
| Преподаватель | Толпинская Н.Б., Строганов Ю.В. |

Москва, 2020 г.

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

На вход 2 списка, проверяется что каждый список является подмножеством другого.

`subsetp` – является предикатом, который возвращает `T`, если каждый элемент списка `list1` встречается в («равен» некоторому элементу в) списке `list2`, иначе `Nil`.

```
1 (defun set-equal (lst1 lst2)
2   (and (subsetp lst1 lst2) (subsetp lst2 lst1))
3 )
```

Примеры:

```
1 (set-equal '(1 4 2 3) '(2 3 1 4))
```

Результат: `T`

```
1 (set-equal '(1 4 3) '(2 3 1 4))
```

Результат: `Nil`

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

Сначала необходимо создать точечные пары.

```
1 (setq countries '(Russia France Germany USA))
2 (setq cities '(Moscow Paris Berlin Vashington))
3
4 (defun c_point (f z)
5   (cons f z)
6 )
7
8 (setq newPoints (mapcar #'c_point countries cities))
```

Результат:

((RUSSIA . MOSCOW) (FRANCE . PARIS) (GERMANY . BERLIN) (USA . VASHINGTON))

Рекурсивный поиск страны по столице и наоборот.

На вход подается страна/столица и список пар. На выходе при успешном поиске возвращается соответствующее значение или `Nil`, если значение не найдено.

```

1 (defun found_country (city lst)
2   (cond ((null lst) nil)
3         ((eql city (cdr (car lst))) (caar lst))
4         (t (found_country city (cdr lst))))
5   )
6 )
7
8 (defun found_city (country lst)
9   (cond ((null lst) nil)
10         ((eql country (car (car lst))) (cdr (car lst)))
11         (t (found_city country (cdr lst))))
12   )
13 )

```

Примеры:

```
1 (found_country 'Moscow newPoints)
```

Результат: RUSSIA

```
1 (found_city 'h newPoints)
```

Результат: NIL

Поиск с использование функционалов.

На вход подается страна/столица и список пар. Определяется функция found, для поиска страны/города. Данная функция будет применяться каскадным образом (к первым двум, затем к результату и следующему и так далее). Следовательно, первый раз первый аргумент будет точечной парой, в последующие атомом, делаем проверку и в зависимости от нее, либо проверяем на совпадение одного из значений точечной пары с аргументом (только в первый раз), либо нет. Второй аргумент проверяется всегда.

```

1 (defun found_country_func (city lst)
2   (defun found (lst1 lst2)
3     (if (consp lst1)
4         (or (if (eql city (cdr lst1)) (car lst1) Nil)
5             (if (eql city (cdr lst2)) (car lst2) Nil) )
6         (or lst1 (if (eql city (cdr lst2)) (car lst2) Nil) )
7     )
8   )
9   (reduce #'found newPoints)
10 )
11
12 (defun found_city_func (country lst)

```

```

13 (defun found (lst1 lst2)
14   (if (consp lst1)
15       (or (if (eql country (car lst1)) (cdr lst1) Nil)
16           (if (eql country (car lst2)) (cdr lst2) Nil) )
17       (or lst1 (if (eql country (car lst2))(cdr lst2) Nil)))
18   )
19 )
20 (reduce #'found newPoints)
21 )

```

Примеры:

```
1 (found_country 'Moscow newPoints)
```

Результат: RUSSIA

```
1 (found_city 'Moscow newPoints)
```

Результат: NIL

3. **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.**

На вход функции список и число, на которое умножать.

С использованием функционалов

Используется функционал `mapcar`, λ -функция (умножение элемента на аргумент, во 2-ой функции сначала проверка на число) применяется к первым элементам списков, затем ко вторым и т.д., и результаты применения собираются в результирующий список.

```

1 (defun multiplication_numbers (lst k)
2   (mapcar #'(lambda (x) (* x k)) lst)
3 )
4
5 (defun multiplication_all (lst k)
6   (mapcar #'(lambda (x) (if (numberp x) (* x k) x)) lst)
7 )

```

Примеры:

```
1 (multiplication_numbers '(1 2 3) 10)
```

Результат: (10 20 30)

```
1 (multiplication_all '(1 2 a 3) 10)
```

Результат: (10 20 A 30)

С использованием рекурсии

Пока список не Nil (каждый раз функция применяется для хвоста списка), с использованием функции cons создается список из обновленных значений.

```
1 (defun mul_numbers_rec (lst k)
2   (if lst
3     (cons (* (car lst) k) (mul_numbers_rec (cdr lst) k))
4   )
5 )
6
7 (defun mul_all_rec (lst k)
8   (if lst
9     (cons
10      (if (numberp (car lst)) (* (car lst) k) (car lst))
11      (mul_all_rec (cdr lst) k)
12    )
13   )
14 )
```

Примеры:

```
1 (mul_numbers_rec '(1 2 3) 10)
```

Результат: (10 20 30)

```
1 (mul_all_rec '(1 2 a 3) 10)
```

Результат: (10 20 A 30)

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

На вход функции список.

С использованием функционалов

Используется функционал mapcar, λ-функция (если число, вычитание 10 из аргумента, иначе оставляем тоже значение) применяется к первым элементам списков, затем ко вторым и т.д., и результаты применения собираются в результирующий список.

```
1 (defun minus_ten (lst)
2   (mapcar #'(lambda (x) (if (numberp x) (- x 10) x)) lst)
3 )
```

Примеры:

```
1 (minus_ten '(1 2 a 3))
```

Результат: (-9 -8 A -7)

С использованием рекурсии

Пока список не Nil (каждый раз функция применяется для хвоста списка), с использованием функции cons создается список из обновленных значений.

```
1 (defun minus_ten_rec (lst)
2   (if lst
3     (cons
4       (if (numberp (car lst)) (- (car lst) 10) (car lst))
5       (minus_ten_rec (cdr lst)))
6     )
7   )
8 )
```

Примеры:

```
1 (minus_ten_rec '(1 2 a 3))
```

Результат: (-9 -8 A -7)

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

На вход функции список.

С использованием функционалов

Используется функционал reduce, определяется функция found, для поиска списка. Данная функция будет применяться каскадным образом (к первым двум, затем к результату и следующему и так далее). Как только находится первый список, немедленно возвращается это значение без проверки остальных.

```
1 (defun first_list (lst)
2   (defun found (lst1 lst2)
3     (or (if (listp lst1) lst1 Nil)
4         (if (listp lst2) lst2 Nil))
5     )
6   )
7   (reduce #'found lst)
8 )
```

Примеры:

```
1 (first_list '(1 (1) (3)))
```

Результат: (1)

С использованием рекурсии

Пока список не Nil (каждый раз функция применяется для хвоста списка), ищется список, если список, то он возвращается.

```
1 (defun first_list_rec (lst)
2   (cond ((null lst) Nil)
3         ((if (listp (car lst)) (car lst)))
4         (t (first_list_rec (cdr lst))))
5   )
6 )
```

Примеры:

```
1 (first_list_rec '(1 (2 3 4) (3)))
```

Результат: (2 3 4)

6. **Написать функцию, которая выбирает из заданного списка только те числа, которые между двумя заданными границами.**

На вход принимаются границы диапазона и список.

С использованием функционалов

Используется функционал reduce, определяется функция found, для проверки вхождения в диапазон. Данная функция будет применяться каскадным образом (к первым двум, затем к результату и следующему и так далее). Следовательно, нужно проверить является ли первый аргумент числом (в первый раз), если так то проверяются оба, нет один. С помощью функции append создается список всех подходящих значений.

```
1 (defun found_between (a b lst)
2   (defun found (lst1 lst2)
3     (if (numberp lst1)
4         (append (if (and (< a lst1) (< lst1 b)) (list lst1) Nil)
5                 (if (and (< a lst2) (< lst2 b)) (list lst2) Nil))
6         (append lst1
7                 (if (and (< a lst2) (< lst2 b)) (list lst2) Nil)))
8   )
9   )
10 (reduce #'found lst)
11 )
```

Примеры:

```
1 (found_between 1 5 '(1 2 3 4 5 6 7 8 9))
```

Результат: (2 3 4)

С использованием рекурсии

Пока список не Nil (каждый раз функция применяется для хвоста списка), ищутся все числа в заданном диапазоне.

```
1 (defun found_between_rec (a b lst)
2   (if lst
3     (append
4       (if (and (< a (car lst)) (< (car lst) b))
5         (list (car lst)) Nil
6       )
7     (found_between_rec a b (cdr lst))
8   )
9 )
10 )
```

Примеры:

```
1 (found_between_rec 1 5 '(1 2 3 4 5 6 7 8 9))
```

Результат: (2 3 4)

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

На вход принимаются 2 списка.

С использованием функционалов

Используются функционалы `mapcar`, `mapcan`, λ -функция (создающая список пробегааясь для каждого X по всем Y) применяется к первым элементам списков, затем ко вторым и т.д., и результаты применения собираются в результирующий список. `mapcar` отличается от `mapcan`: `mapcan` при построении новых данных использует память исходных данных.

```
1 (defun decart (lst1 lst2)
2   (mapcan #'
3     (lambda (x)
4       (mapcar #'(lambda (y) (list x y)) lst2)
5     ) lst1
6   )
7 )
```

Примеры:


```
1 (decart '(1 2 3) '(4 5 6))
```

Результат: ((1 4) (1 5) (1 6) (2 4) (2 5) (2 6) (3 4) (3 5) (3 6))

С использованием рекурсии

Пока список не Nil (каждый раз функция применяется для хвоста списка), происходит формирование списка из головы текущего и всех элементов второго списка (для этого используется 2 рекурсивная функция).

```
1 (defun decart_rec(x y)
2   (cond
3     ((null x) nil)
4     (t (append(second_param(car x) y)(decart_rec(cdr x) y)))
5   )
6 )
7
8 (defun second_param(x y)
9   (cond
10    ((null y) nil)
11    (t (cons(list x (car y)) (second_param x (cdr y))))
12  )
13 )
```

Примеры:

```
1 (decart_rec '(1 2 3) '(4 5 6))
```

Результат: ((1 4) (1 5) (1 6) (2 4) (2 5) (2 6) (3 4) (3 5) (3 6))

8. Почему так реализовано reduce, в чем причина?

(reduce #' + ()) → 0

Обратимся к исходному коду (точнее его части, где описывается функция reduce).

```
1 LISPFUN(reduce, seclass_default, 2, 0, norest, key, 5,
2   (kw(from_end), kw(start), kw(end), kw(key), kw(initial_value)) )
3 { /* (REDUCE function sequence [:from-end] [:start] [:end] [:key]
4   [:initial-value]), CLTL p. 251, CLTL2 p. 397
5   Stack layout: function, sequence, from-end, start, end, key, initial-value. */
6   pushSTACK(get_valid_seq_type(STACK_5)); /* check sequence */
7   /* Stack layout: function, sequence, from-end, start, end, key, initial-value,
8     typdescr. */
9   check_key_arg(&STACK_(1+1)); /* key check */
10  start_default_0(STACK_(3+1)); /* Default value for start is 0 */
11  /* Default value for end is the length of the sequence: */
12  end_default_len(STACK_(2+1), STACK_(5+1), STACK_0);
13  /* check start- and end arguments: */
14  test_start_end(&O(kwpair_start), &STACK_(2+1));
15  { /* subtract and compare start- and end arguments: */
```

```

16   var object count = I_I_minus_I(STACK_(2+1),STACK_(3+1));
17   /* count = (- end start), an integer >=0. */
18   if (eq(count,Fixnum_0)) { /* count = 0 ? */
19     /* start and end are equal */
20     if (!boundp(STACK_(0+1))) { /* initial-value supplied? */
21       /* no -> call function with 0 arguments: */
22       funcall(STACK_(6+1),0);
23     } else {
24       /* yes -> initial-value as result value: */
25       VALUES1(STACK_(0+1));
26     }
27     skipSTACK(7+1);
28     return;
29   }
30   /* common case: start < end, count > 0 */
31   pushSTACK(count);
32 }
33 /* Stack layout: function, sequence, from-end, start, end, key, initial-value,
34                  typdescr, count. */
35 /* check from-end: */
36 ...

```

Остальная часть функции не обрабатывается при наших аргументах, а именно зайдет в условие на 18 строчке и на 28 выйдет. При этом присвоится значение 0, так как значение по умолчанию для суммы (вызов + для 0 аргументов на строчке 22), при заданном начальном значении присвоится оно.

Например:

(reduce #' + () :initial-value 1) → 1

(reduce #' * ()) → 1

Теоретические вопросы:

1. Способы организации повторных вычислений в Lisp.

Использование функционалов или рекурсии.

2. Различные способы использования функционалов.

(mapcar/maplist #'func lst)

mapcar – функция func применяется к головам первым элементам списков, затем ко вторым и т.д., и результаты применения собираются в результирующий список.

maplist – func применяется к “хвостам” списков, начиная с полного списка.

mapscan, mapcon – аналогичны mapcar и maplist, используется память исходных данных, не работают с копиями.

(reduce #'func lst)

reduce – функция func применяется каскадным образом (к первым двум, затем к результату и следующему и так далее).

3. Что такое рекурсия? Способы организации рекурсивных функций.

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

- (a) Хвостовая рекурсия – результат формируется не на выходе из рекурсии, а на входе в рекурсию, все действия выполняя до ухода на следующий шаг рекурсии.
- (b) Рекурсия по нескольким параметрам
- (c) Дополняемая рекурсия – при обращении к рекурсивной функции используется дополнительная функция не в аргументе вызова, а вне его
- (d) Множественная рекурсия – на одной ветке происходит сразу несколько рекурсивных вызовов.

4. Способы повышения эффективности реализации рекурсии.

Использование хвостовой рекурсии. 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.

Преобразование не хвостовой рекурсии в хвостовую, возможно путем использования дополнительных параметров. В этом случае необходимо использовать функцию-оболочку для запуска рекурсивной функции с начальными значениями дополнительных параметров.