Контрольные вопросы к зачету 6 апреля 2023

1. Определите с помощью ламбда-выражения функцию, вычисляющую

$x + y - x*y$.

```
(setq my-func (lambda (x y) (- (+ x y) (* x y))))
(funcall my-func 2 3)
```

2. вычислите значения комплекта ламбда-выражений,

например: ((lambda (x y) (list y x)) 'x y)

Выражение ((lambda (x y) (list y x)) 'x y) вернет список (y x).

- Лямбда-выражение (lambda (x y) (list y x)) принимает два аргумента x и y и возвращает список, в котором эти аргументы идут в обратном порядке, т.е. (list y x).
- Далее, мы вызываем это лямбда-выражение с аргументами 'x и y, т.е. ((lambda (x y) (list y x)) 'x y). В этом случае, 'x привязывается к параметру x, а y привязывается к параметру y. Таким образом, наша функция создаст список из y и x, т.е. (list y x).
- Итоговый результат этого выражения - (y x).

3. Определите функции (NULL X), (CADDR X) и (LIST X1 X2 X3) с помощью базовых функций.

```
(defun my-null (x)
  (eq x nil))

(defun my-caddr (x)
  (car (cdr (cdr x))))

(defun my-list (x1 x2 x3)
  (cons x1 (cons x2 (cons x3 nil))))
```

4. Определите функцию (НАЗОВИ x y), которая определяет функцию с именем, заданным аргументом *x* , и лямбда-выражением *y*. Определите с помощью этой функции функцию, вычисляющую сумму квадратов двух чисел, и саму функцию НАЗОВИ.

```
(defun НАЗОВИ (x y)
  (setf (symbol-function x) y))

(НАЗОВИ 'sum-of-squares
       (lambda (x y)
         (+ (* x x) (* y y))))
(sum-of-squares 3 4)
```

5. Вычислите значения комплекта лямбда-вызовов с ключами, например.

```
((lambda (a b &key c d) (list a b c d)) :a 1 :d 8 :c 6)
```

Выражение ((lambda (a b &key c d) (list a b c d)) :a 1 :d 8 :c 6) вычислится в список (1 NIL 6 8).

Это происходит потому, что данное лямбда-выражение ожидает два обязательных аргумента a и b, а также два необязательных аргумента c и d, которые задаются с помощью ключевых параметров. Ключевые параметры могут быть переданы в любом порядке и могут быть заданы или не заданы.

В данном случае, мы передаем ключевой параметр :a со значением 1, что соответствует обязательному параметру a. Ключевой параметр :d со значением 8 соответствует необязательному параметру d, а ключевой параметр :c со значением 6 соответствует другому необязательному параметру c.

Параметр b не задан явно, поэтому он принимает значение по умолчанию NIL.

Итак, результатом лямбда-выражения будет список (1 NIL 6 8), потому что значение a равно 1, значение b равно NIL, значение c равно 6 и значение d равно 8.

6. В чем различие следующих типов переменных: статические, динамические специальные, глобальные, свободные.

Статические переменные (lexical variables) - это переменные, которые определены внутри лексического блока кода, например, внутри функции или макроса. Они доступны только внутри этого блока и его вложенных блоков, то есть область их видимости ограничена лексическим контекстом. Эти переменные обычно объявляются с помощью специальной формы let или let*.

Динамические специальные переменные (dynamic variables) - это переменные, которые имеют глобальную область видимости, но значение которых может быть изменено в любой момент времени в процессе выполнения программы. Эти переменные обычно объявляются с помощью специальной формы defvar или defparameter. Значение этих переменных хранится в специальной области памяти, называемой динамической областью, и может быть изменено с помощью функции setq.

Глобальные переменные (global variables) - это переменные, которые имеют глобальную область видимости и могут быть доступны в любом месте программы. Они также могут быть изменены с помощью функции setq. Обычно эти переменные объявляются с помощью специальной формы defvar или defparameter.

Свободные переменные (free variables) - это переменные, которые используются внутри функции, но не объявляются внутри этой функции. Они могут быть определены где-то внутри программы или даже вне нее. Значение этих переменных ищется во время выполнения программы в ближайшем контексте, где они были объявлены (внутри функции или глобальной области видимости).

Важно отметить, что специальные и глобальные переменные имеют глобальную область видимости, но значения динамических переменных могут быть установлены локально внутри функции с помощью специальной формы let. Кроме того, лексические переменные могут иметь те же имена, что и динамические и глобальные переменные, но имеют отличную область видимости.

7. Чем отличается статическое вычисление от динамического, если в функции нет свободных переменных?

В Lisp статическое вычисление и динамическое вычисление относятся к механизму связывания символьных значений (symbolic values) и функций (functions) с переменными (variables).

Если в функции нет свободных переменных, то при использовании статического вычисления все символы, используемые в функции, связываются с их значением в момент определения функции, а не в момент ее вызова. Это означает, что значения символов будут зафиксированы в момент определения функции, и даже если значения этих символов изменятся после определения функции, они не будут влиять на работу функции.

Например, рассмотрим следующий код:

```
(defvar x 10)

(defun foo ()
  (print x))

(defun bar ()
  (let ((x 20))
    (foo)))
```

При использовании статического вычисления значение x внутри функции foo будет равно 10, поскольку x было определено вне функции foo и его значение было зафиксировано в момент определения функции. Поэтому вызов функции bar не изменит значение x внутри функции foo.

В динамическом вычислении символы связываются с их текущим значением в момент вызова функции, а не в момент ее определения. Если значение символа изменится после определения функции, это изменение будет отражено в работе функции.

Например, рассмотрим следующий код:

```
(defvar x 10)

(defun foo ()
  (print x))

(defun bar ()
  (let ((x 20))
    (funcall #'foo)))
```

При использовании динамического вычисления значение x внутри функции foo будет равно текущему значению символа x в момент вызова функции. Поэтому вызов функции bar изменит значение x на 20 во время вызова функции foo.

Надеюсь, это поможет вам лучше понять разницу между статическим и динамическим вычислением в Lisp.

8. Определим функции F и G следующим образом:

(defun f(y) (g y))➔F

(defun g(x) (list x y))➔G

Какое значение или сообщение об ошибке будет результатом вызовов функций F и G, например: (f (setq y 'y))?

Вызов функции (f (setq y 'y)) приведет к ошибке, так как переменная y не определена в области видимости функции g.

Для того, чтобы вызов функции f корректно работал, переменная y должна быть определена в текущей области видимости или как аргумент функции f.

9. Запишите следующие ламбда-вызовы с использованием формы LET и вычислите их значения, например, ((lambda (x y) (list x y)) '(+ 2 3) 'c)

```
(let ((x '(+ 2 3))
      (y 'c))
  (list x y))
```

10. С помощью предложения COND или CASE определите функцию, которая возвращает в качестве значения столицу заданного аргументом государства:

(столица 'Финляндия) ➔ХЕЛЬСИНКИ

COND

```
(defun столица (государство)
  (cond ((equal государство 'Финляндия) 'ХЕЛЬСИНКИ)
        ((equal государство 'Россия) 'Москва)
        ((equal государство 'США) 'Вашингтон)
        (t 'Неизвестная столица)))
```

Эта функция принимает один аргумент - название государства и использует cond для проверки его значения и возврата соответствующей столицы. Если государство не найдено в списке, функция возвращает сообщение "Неизвестная столица". В данном случае вызов (столица 'Финляндия) вернет ХЕЛЬСИНКИ.

CASE

```
(defun столица (государство)
  (case государство
    ('Финляндия 'ХЕЛЬСИНКИ)
    ('Россия 'Москва)
    ('США 'Вашингтон)
    (otherwise 'Неизвестная столица)))
```

11. Предикат сравнения (> x y) истинен, если x больше y. Опишите с помощью предиката > и условного предложения функцию, которая возвращает из трех числовых аргументов значение среднего по величине числа: (среднее 4 7 6) ➔ 6

```
(defun среднее (x y z)
  (cond ((and (> x y) (> y z)) y)
        ((and (> y x) (> x z)) x)
        (t z)))
(среднее 7 1 5)
```

12. Можно ли с помощью предложения COND запрограммировать предложение IF как функцию ?

Да, можно использовать cond для определения функции, эквивалентной if.

```
(defun my-if (condition then-clause else-clause)
  (cond (condition then-clause)
        (t else-clause)))
```

Эта функция принимает три аргумента: condition, then-clause и else-clause. Она использует cond для проверки значения condition и возвращает then-clause, если condition истинно, и else-clause в противном случае. Таким образом, эта функция эквивалентна if.

```
(my-if (> 2 1) 'true 'false) ; вернет 'true'
(my-if (< 2 1) 'true 'false) ; вернет 'false'
```

13. Запрограммируйте с помощью предложения DO итеративную версию функции факториал.

```
(defun factorial (n)
  (do ((i n (- i 1))
       (result 1 (* result i)))
      ((<= i 1) result)))
```

Эта функция использует do для вычисления факториала числа n. Она инициализирует две переменные: i со значением n и result со значением 1. Затем, на каждой итерации, она уменьшает значение i на 1 и умножает значение result на i. Цикл завершается, когда i становится меньше или равно 1, и возвращается значение result.

14. Определите функцию (ПРОИЗВЕДЕНИЕ n m), вычисляющую произведение двух целых положительных чисел.

```
(defun ПРОИЗВЕДЕНИЕ (n m)
  (if (or (<= n 0) (<= m 0))
      0
      (* n m)))
```

Функция использует if для проверки того, что оба аргумента n и m являются положительными числами. Если хотя бы один из аргументов не положительный, то функция возвращает 0. В противном случае, функция возвращает произведение n и m.

(ПРОИЗВЕДЕНИЕ 5 3) ; вернет 15

(ПРОИЗВЕДЕНИЕ 0 7) ; вернет 0

(ПРОИЗВЕДЕНИЕ -2 4) ; вернет 0

15. Функция (LENGTH x) является встроенной функцией, которая возвращает длину списка. Определите функцию LENGTH1 сначала рекурсивно с помощью предложения COND и затем итеративно с помощью предложения PROG.

В этой версии COND используется для проверки, является ли список lst пустым. Если это так, то функция возвращает 0. В противном случае функция рекурсивно вызывает себя для остальной части списка cdr lst и добавляет 1 к результату.

```
(defun LENGTH1 (lst)
  (cond ((null lst) 0)
        (t (+ 1 (LENGTH1 (cdr lst))))))
```

Итеративная версия с помощью PROG:

```
(defun LENGTH1 (lst)
  (prog ((count 0)
         (curr lst))
    loop
      (cond ((null curr) (return count)))
      (setf count (+ count 1)
            curr (cdr curr))
      (go loop))))
```

В этой версии PROG используется для создания блока, в котором выполняются итеративные действия. Переменная count используется для подсчета количества элементов в списке, а переменная curr используется для отслеживания текущего элемента списка. Внутри блока loop используется COND для проверки, является ли curr пустым списком. Если это так, то функция возвращает значение count с помощью RETURN. В противном случае функция увеличивает значение count на 1, перемещает curr на следующий элемент списка и переходит обратно на метку loop с помощью GO.

16. Числа Фибоначчи образуют ряд 0, 1, 1, 2, 3, 5, 8 . . . Эту последовательность можно определить с помощью следующей функции FIB:

fib(n) =0, если n=0, fib(n) =1, если n=1, fib(n) =fib(n-1)+fib(n-2), если n>1

Определите лисповскую функцию fib(n), вычисляющую n–й элемент ряда Фибоначчи.

```
(defun fib (n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (t (+ (fib (- n 1)) (fib (- n 2))))))
```

В данной функции мы используем условное выражение cond для определения случаев, когда n=0, n=1 или n>1. Если n=0, то возвращается 0, если n=1, то возвращается 1. Если n>1, то мы вызываем функцию fib для n-1 и n-2 и возвращаем их сумму.

17. Определите функцию ДОБАВЬ, прибавляющую к элементам списка данное число:

(добавь '(2 7 3) 3)➔(5 10 6).

```
(defun добавь (lst num)
  (if (null lst)
      '()
      (cons (+ (car lst) num)
            (добавь (cdr lst) num))))
```

Здесь используется рекурсивная функция, которая обрабатывает список lst. Если список пуст, то возвращается пустой список. Иначе, к первому элементу списка lst прибавляется число num, и результат добавляется в начало списка, после чего функция рекурсивно вызывается для оставшейся части списка lst.

```
(добавь '() 10) ; Результат: ()
(добавь '(1) 0) ; Результат: (1)
(добавь '(-1 0 1) 2) ; Результат: (1 2 3)
```

18. Напишите функцию читай-фразу, которая вводит фразу на естественном языке, заканчивающуюся вопросительным или восклицательным знаком, и преобразует ее в список, например : (куда дует ветер ?)

Для решения задачи можно использовать функцию read-line, чтобы считать строку с консоли, а затем разбить ее на слова с помощью функции split-string. Далее можно использовать цикл do для прохода по списку слов и удаления знаков препинания с помощью функции substitute.

```
(defun читай-фразу ()
  "Вводит фразу на естественном языке и возвращает ее в виде списка слов."
  (let ((фраза (split-string (read-line) " ")))
    (do ((i 0 (1+ i)))
```

```
      ((>= i (length фраза)) фраза)
    (setf (nth i фраза)
          (substitute nil '(#\? #\!) (elt (nth i фраза) (1- (length (nth i фраза))))))))))
(читай-фразу)
;; Ввод: "Куда дует ветер?"
;; Вывод: ("Куда" "дует" "ветер")
```

19. Запрограммируйте функцию читай, которая принимает вызов функции в виде

fn(*a1 a2 . . . aN*) и возвращает значение такого вызова, например

(читай + (2 3))➔ 5

```
(defun читай (операция &rest аргументы)
  (apply операция аргументы))

(читай '+ 2 3) ; returns 5
(читай '- 5 2) ; returns 3
(читай '* 2 3) ; returns 6
(читай '/ 6 2) ; returns 3
```

20. Определите функцию, которая создаст на выходе прямоугольник с заданными сторонами *м* и *n*), иззвездочек (*).

```
(defun rectangle (m n)
  (let ((line (make-string n :initial-element #\*)))
    (dotimes (i m)
      (format t "~a~%" line))))
```

В этой функции используется форма make-string, чтобы создать строку длиной n, состоящую из символов *. Затем мы повторяем эту строку m раз с помощью цикла dotimes и выводим каждую строку на новой строке с помощью format.

```
(rectangle 5 10)
; **********
**********
**********
**********
**********
```

21. Используя функцию *черта* , напишите функцию

(прямоугольник *n m*) заполняющую всю область *n x m* звездочками.

```
(defun прямоугольник (n m)
```

```
 (let ((line (make-string m :initial-element #\*)))
   (dotimes (i n)
     (format t "~a~%" line))))
```

22 Напишите программу, которая создаст на выходе прямоугульник из звездочек.

```
(defun draw-rectangle (width height)
  (dotimes (i height)
    (dotimes (j width)
      (format t "*"))
    (format t "~%")))
```

23. Определите функцию, которая спрашивает у пользователя имя и вежливо отвечает:

Введите ваше имя: Михаил

Прекрасно, Михаил, программа работает!

```
(defun greet-user ()
  (format t "Введите ваше имя: ")
  (let ((name (read-line)))
    (format t "Прекрасно, ~a, программа работает!" name)))
```

Эта функция использует функцию format, которая позволяет форматировать вывод. Она выводит строку "Введите ваше имя: ", затем читает строку, введенную пользователем, с помощью функции read-line, и сохраняет ее в переменную name. Затем функция с помощью format выводит приветствие, вставляя имя пользователя вместо символа ~a.

24 Напишите программу на Лиспе, запрашивающую имя и затем вежливо обращаясь к нему или ей в зависимости от возраста, граница которого 18т лет.

```
(defun greet-user ()
  (format t "Введите ваше имя: ")
  (let ((name (read-line)))
    (format t "Сколько вам лет, ~a? " name)
    (let ((age (parse-integer (read-line))))
      (cond ((< age 18)
             (format t "Привет, ~a! Ты еще слишком молод для этой программы." name))
            ((>= age 18)
             (format t "Здравствуйте, ~a! Рады видеть вас в нашей программе." name))))))
```

(greet-user)

Введите ваше имя: Анна

Сколько вам лет, Анна? 20

Здравствуйте, Анна! Рады видеть вас в нашей программе.

25. Вычислите (напишите) программу значения следующих вызовов функции funcall, например: (funcall 'list '(a b))

Вызовы функции funcall можно использовать для вызова функций, передавая аргументы в виде списка. Например, (funcall 'list '(a b)) вернет список из элементов a и b, то есть '(a b).

1- (funcall 'cons 'a 'b) вернет список из элементов a и b, то есть '(a . b).
2- (funcall 'list 'a 'b) вернет список из элементов a и b, то есть '(a b).
3- (funcall '+ 2 3) вернет сумму чисел 2 и 3, то есть 5.
4- (funcall 'string 'a 'b) вернет строку, состоящую из символов a и b, то есть "ab".

Функция funcall позволяет вызывать любую функцию, даже если ее имя хранится в переменной.

26. Определите функционал (MAPLIST fn список) для одного списочного аргумента. Например, для списка (1 2 3) и функции (lambda (x) (+ x 1)), MAPCAR вернет (2 3 4), а MAPLIST вернет ((2 3 4) (3 4) (4)), т.к. он применяет функцию к первому подсписку (1 2 3), затем ко второму подсписку (2 3), и, наконец, к последнему элементу (3).

Функционал (MAPLIST fn список) применяет функцию fn к каждому элементу списка и возвращает список результатов. Однако, в отличие от функционала MAPCAR, который применяет функцию только к элементам списков аргументов и возвращает список, состоящий из результатов, MAPLIST применяет функцию ко всем подспискам исходного списка, начиная с первого, и возвращает список, состоящий из результатов, начиная с первого.

```
(defun maplist (fn lst)
  (cond ((null lst) '())
        (t (cons (apply fn lst) (maplist fn (cdr lst))))))
(maplist (lambda (x) (* x x)) '(1 2 3 4)) ; вернет ((1 4 9 16) (4 9 16) (9 16) (16))
```

Здесь функция (lambda (x) (* x x)) применяется ко всем подспискам списка '(1 2 3 4)', начиная с первого, и возвращает список, состоящий из результатов, начиная с первого.

27.Определите функционал (APL-APPLY *f x*), который применяет каждую функцию *fi* списка *f*=(*f1, f2, . . . , fN*) к соответствующему элементу *xi* списка *x*=(*x1, x2, . . . , xN*) и возвращает список, сформированный из результатов.

```
(defun APL-APPLY (f x)
```

```
  (mapcar (lambda (ff xx) (funcall ff xx)) f x))
```

Эта функция принимает два аргумента: список функций f и список значений x. Затем она применяет каждую функцию из списка f к соответствующему элементу из списка x с помощью функции mapcar, которая возвращает список результатов. Функция funcall используется для вызова каждой функции из списка f с соответствующим элементом из списка x.

```
(defun square (x) (* x x))
(defun double (x) (+ x x))

(setq funcs '(square double))
(setq nums '(1 2 3))

(print (APL-APPLY funcs nums)) ; (1 4 9 2 4 6)
```

28. Определите функциональный предикат (КАЖДЫЙ *пред список*), который истинен в том и только в том случае, когда являющийся функциональным аргументом предикат *пред* истинен для всех элементов списка *список*.

```
(defun каждый (pred список)
  (cond ((null список) t)
        ((funcall pred (car список)) (каждый pred (cdr список)))
        (t nil)))
```

Эта функция принимает два аргумента: предикат и список. Она рекурсивно проходит по каждому элементу списка и проверяет, удовлетворяет ли элемент предикату. Если все элементы удовлетворяют предикату, то функция возвращает true, иначе false.

29. Определите функциональный предикат (НЕКОТОРЫЙ *пред список*), который истинен, когда предикат истинен хотя бы для одного элемента списка.

```
(defun некоторый (pred список)
  (cond ((null список) nil)
        ((funcall pred (car список)) t)
        (t (некоторый pred (cdr список)))))
```

Эта функция принимает предикат и список в качестве аргументов и рекурсивно проверяет каждый элемент списка. Если предикат истинен для какого-либо элемента списка, функция возвращает true. Если все элементы списка были проверены, и предикат не был истинным ни разу, функция возвращает false.

30. Определите фильтр (УДАЛИТЬ-ЕСЛИ *пред список*),

удаляющие из списка элементы, которые обладают или не обладают свойством, наличие которого проверяет предикат *пред*.

```
defun УДАЛИТЬ-ЕСЛИ (пред список)
  (cond ((null список) '())
        ((funcall пред (car список))
         (УДАЛИТЬ-ЕСЛИ пред (cdr список)))
        (t (cons (car список) (УДАЛИТЬ-ЕСЛИ пред (cdr список))))))

; Примеры использования
(УДАЛИТЬ-ЕСЛИ #'oddp '(1 2 3 4 5)) ; вернет (2 4)
(УДАЛИТЬ-ЕСЛИ #'stringp '(1 "two" 3 "four")) ; вернет (1 3)
```

Для определения фильтра УДАЛИТЬ-ЕСЛИ можно использовать рекурсивный подход и проверять каждый элемент списка на соответствие предикату пред. Если элемент удовлетворяет условию, то он оставляется в списке, а если нет, то он удаляется.

В данном примере функция УДАЛИТЬ-ЕСЛИ принимает два аргумента: предикат (переменная пред), проверяющий условие, и список (переменная список), из которого нужно удалить элементы. Функция рекурсивно обходит список и проверяет каждый его элемент на соответствие предикату. Если элемент не удовлетворяет условию, то он сохраняется в результирующем списке, иначе он пропускается. Результатом работы функции является список элементов, прошедших проверку предикатом.

31. Напишите программу на Лисп рекурсивного вычисления n!

```
(defun factorial (n)
  (if (<= n 0) 1
      (* n (factorial (- n 1)))))
```

Эта функция проверяет, равен ли аргумент 0 или отрицательному числу. Если да, то функция возвращает 1, иначе она умножает аргумент на результат вызова функции для предыдущего числа. Эта рекурсивная функция будет продолжаться, пока аргумент не станет равным 0, тогда она вернет 1.

32. Напишите программу на Лиспе рекурсивного вычисления “e в степени x”

```
(defun e-to-x (x n)
  (if (= n 0)
      1
      (+ (/ (expt x n) (factorial n))
         (e-to-x x (- n 1)))))
```

```
(defun factorial (n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))

(defun e (x)
  (e-to-x x 9))
```

Функция e-to-x вычисляет значение e^x для заданного x и количества итераций n. Мы используем формулу Тейлора для приближенного вычисления e^x:

e^x = 1 + x/1! + x^2/2! + ... + x^n/n!

Функция factorial вычисляет факториал числа, который используется в формуле Тейлора.

Функция e вызывает e-to-x с количеством итераций, равным 9, чтобы получить более точный результат.

```
(e 2) ;; Output: 7.3890560989306495
```