LISP
язык символичной обработки (не счетный язык)
символ - какой-либо знак
символ - последовательность символов без пробелов

не типизированный язык (можно обойтись без типов данных)

символьные атомы

структуры в паскале: (подумать до следующей недели)
список -

переменная - символьная ссылка на область памяти, которая используется в программе
структура данных в паскале: массивы (совокупность элементов одного типа), ….

список - динамическая структура данных, которая может быть как пустым, так и не пустым
понятие рекурсия - ссылка при описании объекта на сам описываемый объект
список описывается рекурсивно

самый элементарный элемент в лиспе - атом
есть атом и есть список (более сложна конструкция)

как это организовать в памяти?

(A .B) - 2 атома - точечная пара

непустой список - частный случай непустой пары (голова, все остальное), если 2й элемент этой пары список, то это список, если не список, то это точечная пара

список - точечная пара у которой 2й элемент является списком

(A.(B.(c.Nil))) - список
(ABC) - современная запись списка
() - Nil (особый атом, который может обозначать пустой список)
(A) ~ (A.Nil)

A - car - указатель на голову
B - cdr - указатель на все остальное

списковая ячейка - представляет указатель на голову и на хвост
голова может иметь любую природу (список, атом,…)

одноуровневый список
структурированный список (получается дерево)

лист предложил символьный вычисления
единый способ представления (с помощью списков)
на синтаксическом уровне отличия данных от программы нет
в процессе работы программа может править сама себя и далее продолжать работать по новому условию
тогда надо понимать: где данные где программа

по умолчанию функция всегда считает, что первый элемент функции это ее имя, а все остальное это аргументы
z = x + y
(A B C)
A - имя функции BC - аргументы

car переходит по указателю получает доступ к голове
cdr переходит по указателю получает доступ к списку

caadr - car car cdr

атомы, точные пары, списки
список частный случай точной пары

предикаты - логическая функция (функция которая имеет логическое значение, значение функции зависит от значения оргумента)
высказывания - логическая константа

‘(A(BC)) ‘ говорит о том, что вычисления блокируются, т.е. считает что это данные

car - переход по указателю (голова)

чистые функции - функции у которых фиксированное количество аргументов и один результат

в листе любая функция возвращает 1 результат

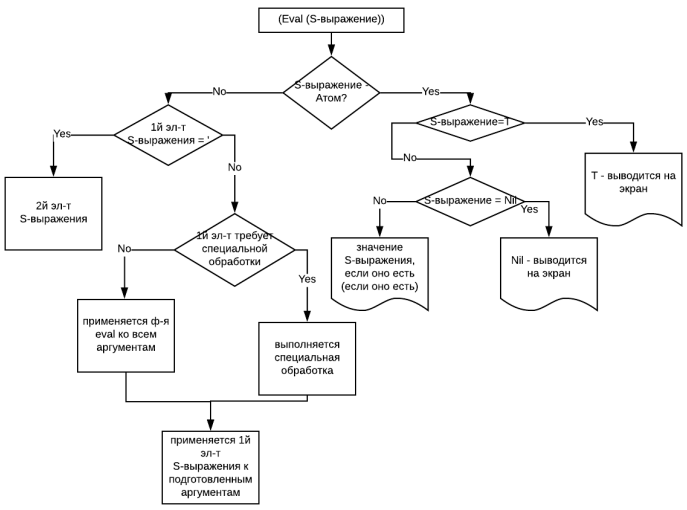
(A’(BC)) - A с аргументами BC
(A(BC)) - A с аргументом B, B с аргументом C

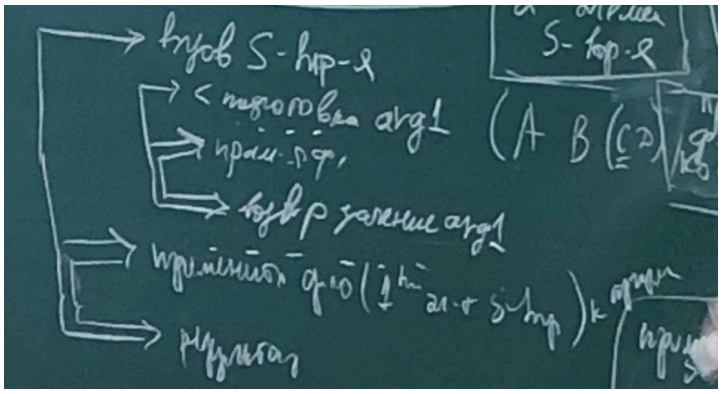
(+1235) - имя функции + и аргументы 1235 , результат - 11

конструкторы (3 лабораторная работа)
cons - базисная функция с фиксированным количеством аргументов
list - не базисная функция, организованна на базе const

функции более высоких порядков - когда в качестве аргумента передается функция

common lisp - установить





кафедра
предмет

вопросы лабораторная №1
1. базовые элементы языка, их определения
2. внутреннее представление списка
3. назначение списка (список используется для представление данных и программы), назначение ‘
4. как система трактует элемент списка

вопросы лабораторная работа №2
1. как выполняются функции car и cdr, как они выполняются, какие результаты вернут, если применить их к 1му номеру

11.02.2019

Список - это частный случай S-выражения, который может быть пустой или непустой . Если он не пустой, то представляет собой первый элемент S-выражение, 2й элемент - список.

Базис Лиспа:
- Атомы
- Структуры (бинарные узлы)
  - Базисные функции
    - atom
    - eq
    - cons
    - car
    - cdr
    - quote ~ ‘
    - cond
    - lambda
    - label
    - eval
  - Базисный функционал

Над базисом строятся простые формулы, где первый элемент виде функции, а остальные - ее аргументы. Все остальные вычисления и преобразования могут быть приведены к этому списку.

Классификация функций:
1. «Чистые» (строго математические функции) (с фиксированным количеством аргументов)
2. специальные функции (способны обрабатывать аргументы нестандартным способом)
3. псевдо функции (осуществляют побочный эффект на аппаратуре)
4. функции, допускающие вариантные значения (позволяют реализовывать логическое программирование)
5. функции высших порядков (используются для синтаксически управляемого конструирования программ)

в современных диалектах лиспа допускаются «ленивые вычисления» - сведения вызовов функций к представлению рецепта (правила) их вычисления (вычисления осуществляется)

Классификация базисных функций чистых (чистые - одноаргументные функции представляющиеся структурами (список?)):
1. Селекторы
   1. car
   2. cdr
2. конструкторы (позволяют создавать структуры)
   1. cons (создает списковую ячейку) (имеет 2 аргумента) (базисная функция)
      1. (cons ‘A ‘B) -> (A. B)
      2. (cons ‘A ‘(B)) -> (A B)
   2. list (создает список) (не базисная функция - не чистая) (можно использовать произвольное количество аргументов) (всегда создает список, нельзя создать точечную пару) (образует столько списковых ячеек, сколько аргументов, последний указатель устанавливается в Nil)
3. Предикат (логическая функция) (все, что не Nil - это TRUE)
   1. atom
   2. consp (проверяет, состоит ли структура из списковых ячеек)
   3. listp (является ли структура списком)
   4. null (пустой список или нет)
   5. numberp (числовое значение или нет)
   6. oddp (проверяет на нечетность)
   7. evenp (проверяет на четность)
   8. eq (2 аргумента) (сравнение указателей) (применима только к символьным атомам) (не может сравнивать списки)

9. eql (делает все что и eq + сравнивает числа одного и того же «типа» (формы представления))
    1. (eql 3.0 3.0) -> T
    2. (eql 3.0 3) -> Nil
10. =
    1. (= 3.0 3) -> T
11. equl ~ eql+списки
12. equlp (работает долго, но проверяет все и атомы и списки)

во всех стандартных функций в качестве встроенных используется функция eql
если нужно сравнивать списки, то нужно это делать руками

ВЫЧИСЛЕНИЕ ФУНКЦИЙ И ВЫПОЛНЕНИЯ ПРОГРАММ НА LISP
СПОСОБЫ ОПРЕДЕЛЕНИЯ ФУНКЦИЙ

(setf name value) - установить символьному атому name значение value

ОПРЕДЕЛЕНИЕ ФУНКЦИЙ
(defun name (arg1, arg2) (форма))

форма - s выражение по которому работает функция

(defun average (x y) (/ (+ x y) 2.0))

λ-нотация функции (способ определения функции без имени)
(lambda (<список аргументов>) (<форма>))
(lambda (x) (+ (* x x x) (* 3 x)))

позволяет определить новую функцию, у которой не будет имени
список аргументов (разделяются пробелом) блокировать нельзя

вызывается с помощью специальной функции
(apply #' average '(5 7)) -> 6.0

если перед (5 7) не поставить ', то 5 будет восприниматься как имя функции

' - блокировка функциональных вычислений
#' - функциональная блокировка (без пробела - один символьный атом) (стоит перед функцией)

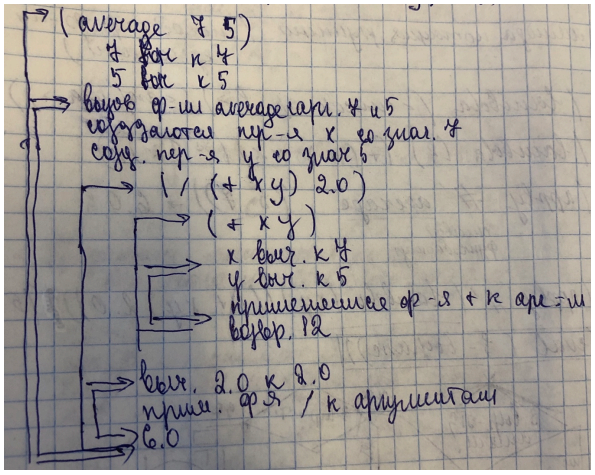
(apply #' (lambda (x y) (/ (+ x y) 2.0)) '(5 7))

КАК РЕАЛИЗУЕТСЯ ПРОГРАММА?
eval - запускается всегда автоматически, когда нажимается ввод
автоматически создается s-выражение бла бла бла

(Eval (S-выражение))

18.02.2019

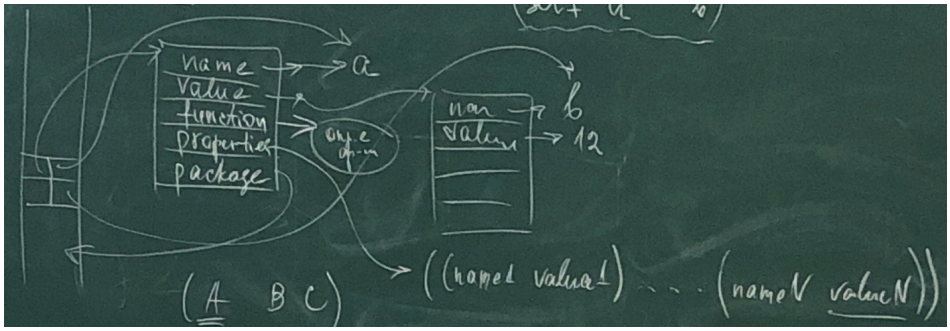
(defun average (x y) (/ (+ x y) 2.0))



ПРЕДСТАВЛЕНИЕ В ПАМЯТИ СИМВОЛЬНЫХ АТОМОВ

символьный атом представляется 5ю указателями
1.указатель на имя
2.указатель на значение
3.указатель на функцию (одновременно атом может быть связан со значением и связан определением функции)
4. указатель на свойство (свойство это структура состоящая из двухэлементных списков)
5. указатель на пакет (список атомов (динамическая структура) , доступных в одном сеансе работы)

символьный атом который связывает со значением с помощью функции setf - это глобальный символьный атом
система не позволит создать второй такой символьный атом, но можно меня его значение



возникает ссылка на символьный атом и одновременно с этим возникает ссылка на 5 указателей

(self a ‘b)

СПЕЦИАЛЬНЫЕ СТАНДАРТНЫЕ ФУНКЦИИ (формы)

базисная функция COND

специальные функции отличаются тем, что в этих функциях либо произвольное количество аргументов, либо не все аргументы обрабатывает (может быть и то и то)

(cond (test1 body1<какая-то форма вместо body1>)
(test2 body2)

…
(testn bodyn))

если все тесты возвращают Nil, то cond вернет Nil

в качестве последнего выражения можно написать T
(T bodyn)

на базе этой функции созданы другие функции

(if test T_body F_body) - функция (не все аргументы будут обработаны)

(not S-выражение) - все что не Nil, то T (и наоборот)

(and arg1 arg2 … argn) - может быть произвольное количество аргументов (обрабатывает не все аргументы, если хотя бы один аргумент = Nil, то системе очевидно какой результат, и не будет дальше обрабатывать аргументы)

(or arg1 arg2 … argn) - работает как и and

пример к лабе:
(defun how_alike (x y) (cond ((or (= x y) (equal x y)) 'the_same) ((and (oddp x) (oddp y)) ' both_odd) ((and (evenp x) (evenp y)) 'both_even) (T 'difference)))

лиспе есть функции - let-формы , которые позволяю предварительно установить каким-то символьным атомом установить значение (это локальная связь, после того как все выполним, они освободят память) (2 аргумента: 1й-список, 2й-тело)

(let ((var1 value1)
(var2 value2)
…
(varn valuen))
body)

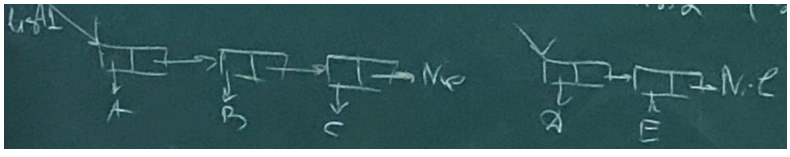
эти символьные атомы выступают в роли локальных переметных (не вносятся в символьный пакет)

(сначала готовятся заданные аргументы, вычисляются выражения value)
связывание значений var с соответствующим значением производят в произвольном порядке
поэтому нельзя ссылаться на значение переменных, Так как в этот момент переменная может указывать на другое значение

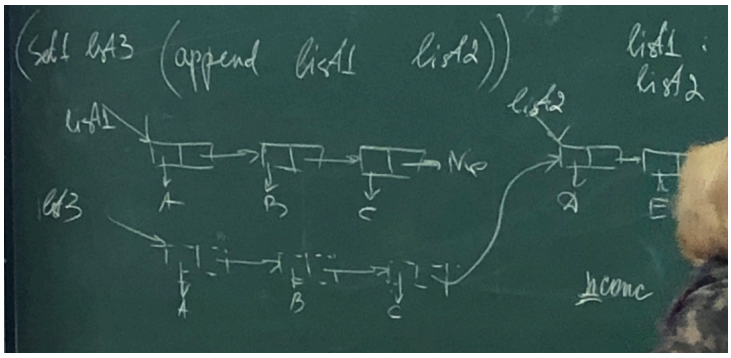
let* - обрабатывает аргументы в жестком порядке, поэтому работает медленнее, чем let

ОПЕРАЦИИ СО СПИСКАМИ
1. структура разрушающие
2. не разрушающие структуру (работают медленнее)

(append list1 list2) - объединяет списки (многоаргументная)
list1: (A B C)
list2: (D E)



создает копии всех аргументов кроме последнего
на это копирование уходит время



если она создает такие копии то остается возможность работать со списком1 со списком2 со списком3
если меняем список3 то меняется список2

nconc - не создает копии (теряется возможность работать со списками отдельно)

reverse - обращает список (одноаргументна) (не может обрабатывать атом) (работает с копией) (разворачивают указатели только верхнего уровня списка)

nreverse - не создает копию (работает по верхнему уровню спусковых ячеек)

last - система ищет последнюю спискровую ячейку верхнего уровня

nth - голову n-й списковой ячейки верхнего уровня (нумерация с 0)

nthcdr - остаток списка начиная n-го списковой ячейки верхнего уровня (нумерация с 0)

все стандартные функции работают только по верхнему уровлю списка

lengs - количество ячеек верхнего уровня списка (может работать с любой структурой, которая в памяти представляется как спусковая ячейка)

удаляет один из элементов списка
(remove el lst)
(remove ‘a ‘(bac)) -> (bc)
(remove ‘(a b) ‘((a b) c d)) -> ((a b) c d)

если надо удалить исходный структурированный список, то система должна использовать в качестве сравнения не eq, а equal
(remove ‘(a b) ‘((a b) c d): test #’equal) -> (c d)

test - используется для передачи имени параметра

member - проверяет присутствует ли элемент в списке (возвращает Nil или остаток списка, начиная с искомого элемента) (проходит только по верхнему списку ячеек сравнивает car указатели)
(member el spis)

внутри использует eq, надо учитывать, что иногда придется переделать на equal

rplaca - замена по указателю car (replaca lst el)
rplacd - замена указателя cdr (replacd lst el)

берется указатель и переставляется
искать ничего не надо => не надо сравнивать

4.03.2019

РАБОТА СО СПИСКАМИ КАК С МНОЖЕСТВАМИ
РАБОТА С АССОЦИАТИВНЫМИ ТАБЛИЦАМИ

список - способ организации дааннных
со списками можно работать как со множествами
но не со всем списками

для множества работает member

существуют стандартные функции, которые объединяют 2 списка, которые являются множествами
unite

ассоциативные таблицы представляются в виде списков точечных пар, которые воспринимаются как ключ-значения
((key1 . value1) (key2 . value2) …)

существуют стандартные функции, которые позволяют по ключу найти значение или наоборот
(assoc key table) - ко ключу возвращается точечная пара (вся списковая ячейка)

(rassoc value table) - по значению возвращается точечная пара

таблица имеет аналогичную списку организацию памяти
таблица - особая структура (структурированный список)

используя списки, можно организовать любые динамические структуры (стеки, очереди, ….)

ФУНКЦИОНАЛЫ
в качестве функционального объекта можно использовать функцию с именем или лямбда-определение
1. применяющие функционал
    1. (apply #'fun '(arg1 … argn)) (#' аналог function)
    2. (funcall #'fun 'arg1 'arg2) - является обратной по отношению к функциональной блокировки (funcall уничтожает функциональную блокировку) (аргументы подаются не списком)
2. отображающий функционал - в качестве аргумента - функциональный объект (функционал использует переданную функцию многократно)
    1. (mapcar #'fun list1) - функцию fun применяет отдельно к каждому элементу списка list1, результат - множество из результатов (проход по верхнему уровню спусковой ячейки) (функция одногоаргументая - аргумент список)
    2. (mapcar #'fun lst1 lst2) - работа заканчивается, когда заканчивает короткий список, результат объединяется в 1 список
    3. (mapcar #'(lambda (x) (* x x)) '(1 2 3 4)) - умножает каждый элемент сам на себя результат -> (1 4 9 16)
    4. (maplist #'fun lst) - объект fun применяется многократно сначала ко всему списка lst, потом к хвосту этого списка, потом к хвосту хвоста и т.д. результаты объединяются в один список
        1. (maplist #'reverse '(a b c)) -> ((a b a) (c b) (c))
        2. (maplist #'fun lst1 lst2) - получается множество результатов которые объединяются в общий список
    5. (find-if #' predicat lst) - возвращает первый элемент списка удовлетворяющий данному предикату или NIL
    6. (remove-if #'predicat lst) - удаляет элементы
    7. (reduce #'fun lst)- позволяют функцию fun применять каскадно к списку (fun должна быть двухаргументна)
        1. (reduce #'+ '(1 2 3 4)) -> 10 (1+2, 3+4, 3+7)

mapcar и maplist - работают с копиями
mapcan (mapcar) и mapcon (maplist) - не создается копия

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ ФУНКЦИОНАЛОВ
(defun decant (lstX lstY)

```
(mapcan #'(lambda (x) (mapcar #'(lambda (y) (list x y)) lstY)) lstX))
```

(decart '(a b) '(1 2)) -> ((a 1) (a 2) (b 1) (b 2))

—-

```
(defun consist-of (lst) (if (member (car lst) (cdr lst) 1 0)))

(defun all_last_element (lst) (if (eql (consist-of lst) 0) (list (car lst) ()))

(defun collection_to_set (lst) (mapcon #'all_last_element lst))
```

(collection_to_set '(i t i g t k s i f k)) -> (g t s i f k) - последнее вхождение каждого элемента

РЕКУРСИЯ

вся информация хранится в куче
эффективность рекурсии - вопрос реализации
в лиспе рекурсия используется достаточно широко
списки организованы рекурсивным описанием

1. простая рекурсия
2. рекурсия 2го порядка - рекурсия несколько раз
3. взаимная рекурсия - описывается несколько функций которые рекурсивно вызывают друг друга

3 проблемы при организации функции:
1. когда остановить рекурсию, чтобы получить промежуточный результат?
2. как начать
3. ????

самый простой вид рекурсии - 1 вызов внутри рекурсии: хвостовая рекурсия

рекурсивные функции чаще всего выполняется с помощью функции cond

```
(defun fun (x)
        (cond (end_test end_value) - условий выхода может быть несколько, в разных случаях разные результаты
                (t (fun (changed_x))))) - все действия происходят с новыми значениями
```

ДОПОЛНЯЕМАЯ РЕКУРСИЯ

при обращении к рекурсивной функции используется дополнительная функция, но не в качестве аргумента вызова, а вне его

```
(defun fun (x)
        (cond (end_test end_value)
                (t (add_function add_value
                                (fun (changed_x))))))
```

пример

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

———

```
(defun fn (x)
	(cond (end_test end_value)
		(t (cons new_el
			(fn (changed_x))))))
```

— — —

```
(defun fn (x)
	(cond (end_test end_value)
		(add_test (add_function add_value
				(fn (changed1_x))))
		(t (fn (changed2_x)))))
```

— — —

```
(defun fn (x)
	(cond (end_test end_value)
		(t (combiner (fn (changed1_x))
				(fn (changed2_x))))))
```

ф-я которая комбинирует 2 рекурсивных вызова

хвостовая рекурсия
для преобразования нехвостовой рекурсии в хвостовую, рекомендуется использовать дополнительный параметр, в котором постепенно будет формироваться результат работы рекурсивной функции
т.о. в описании функции возникает аргументов больше, чем требуется пользователю
очень часто 1й вызов рекурсивной функции выполняют с начальным значением параметра, предназначенного для результата равным либо пустому списку, либо нулем, если это числовой результат или с фиксированным числовым значением

18.03.19

---

1. Найти длину списка

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

2. Сортировка методом вставки
добавление 1го элемента

```
(defun insert_hlp(x lst)
	(cond ((null lst) (list x))
```

```
                ((<= x (car lst)) (cons x list))
                (t (cons (car lst) (insert_hlp x (cdr lst))))))
```

добавление списков

```
(defun sort_help(lst1 lst2)
        (cond ((null lst1 lst2) lst2)
                (t (sort_help (cdr lst1) (insert_hlp (car lst1) lst2)))))

(defun sort_ins(lst)
        (sort_help lst ()))
```

3. дан список из которого необходимо выделить элементы

```
(defun extract_symb(lst)
        (cond ((null lst) ())
                ((symbolp (car lst) (cons (car lst) (extract_symb (cdr lst)))))
                (t (extract_symb (cdr lst)))))
```

4. Первое число

```
(defun first_numb(lst)
        (cond ((numberp lst) lst)
                ((atom lst) nil)
                (t (or (first_numb (car lst)) (first_numb (cdr lst))))))
```

5. Функция, которая подсчитывает количество всех ячеек на каждом уровне

```
(defun cons_sells(lst)
        (if (atom lst) 0
                (t (length lst) (reduce #'+ (mapcar #'cons_sells lst))))))
```

6. Функция, котора преобразует структурированный список в одноуровневый

```
(defun into_one (lst rst)
        (cond ((null lst) rst)
                ((atom lst) (cons lst rst))
                (t (into_one (car lst) (into_one (cdr lst) rst)))))



(defun fun ())
```