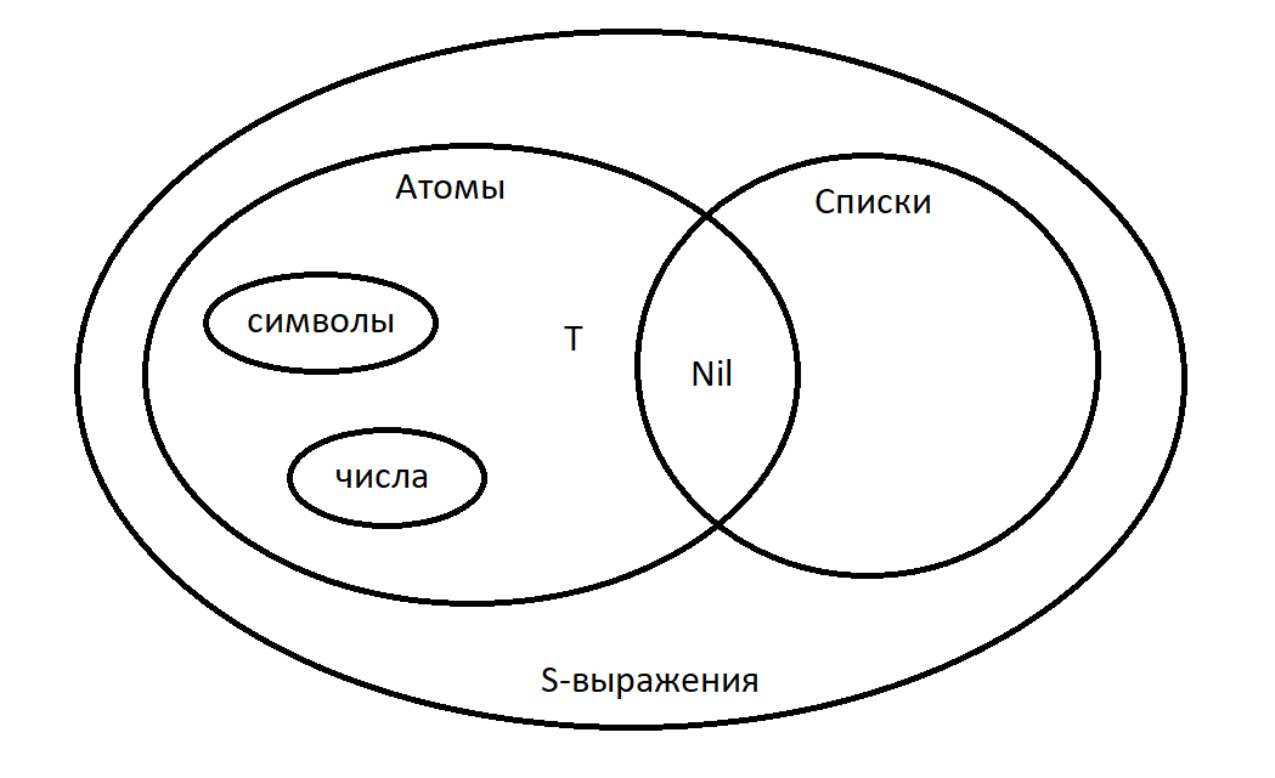
**LISP** - List Processing(обработка списка)

**Функциональный язык** - тот, который базируется на понятии функции.

Вся информация в Lisp представляется в виде символьных выражений. S-выражение - это атом или точечная пара. S потому что символ.



**Особенность языка:** все(программа и данные) в Lisp являются списком.

**Список** - динамическая структура данных. Может быть пустой и непустой. Если непустой список, то он состоит из двух элементов: первый(голова) - любой структуры(нет слова тип), а второй(хвост) - список.

По умолчанию первый аргумент списка - имя функции, остальные - аргументы функции. Заблокировать вычисления можно при помощи ф-и quote.

**' или (quote)** - функция, блокирующая вычисления. Нужна для того, чтобы отличить программу от данных.

**Базис языка** - минимальное средство, которое присутствует в языке.

**Базис** - это минимально необходимый набор конструкций с помощью которого можно написать программу.

**Базисные функции** - предоставляются языком, минимально необходимые для работы языка.

**Элементы языка(базис):**

* атомы
* структуры(список/точечная пара)
* базовые функции
* базовые функционалы

# **Атомы в Lisp**

* символы(идентификаторы) - набор литер (буквы и цифры латинского алфавита);
* самоопределимые атомы - те символы, значения которых знает интерпретатор(числа и символы в двойных кавычках (строки));
* специальные символы - T и Nil (логические константы).

В памяти атомы представлены по разному.

**Символьный атом представляется 5ю указателями:**

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

**Предикат** (логическая функция) (все, что не Nil - это TRUE)

* + atom
  + consp (проверяет, состоит ли структура из списковых ячеек)
  + listp (является ли структура списком)
  + null (пустой список или нет)
  + numberp (числовое значение или нет)
  + oddp (проверяет на нечетность)
  + evenp (проверяет на четность)
  + eq (2 аргумента) (сравнение указателей) (применима только к символьным атомам) (не может сравнивать списки)
  + eql (делает все что и eq + сравнивает числа одного и того же «типа» (формы представления))

**(eql 3.0 3.0) -> T**

**(eql 3.0 3) -> Nil**

* + =

**(= 3.0 3) -> T**

* + equl ~ eql+списки
  + 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

1. **Классификация функций:**

* чистые математические функции имеют фиксированное количество аргумента и возвращают всегда один результат
* рекурсивные
* специальные функции(формы) имеют переменное количество аргументов, не все одинаково обрабатываются
* псевдофункции создают какой-то эффект
* функции с вариантными значениями(выбирается только один результат)
* функции высших порядков (когда аргумент является функцией) используются для создания синтаксически управляемых программ(искусственный интеллект)
* функции селекторы (car, cdr)
* функции конструкторы (cons-базисная, list-произвольная)
* функции предикаты (null, atom, listp, consp)
* функции сравнения (eq, eql, equal, equalp)

1. **Базис - минимальное средство, которое присутствует в языке.**

Базис состоит из:

* атомов
* структур
* базовых функций
* базовых функционалов

1. **Реализация car и cdr**

Функции car и cdr являются базовыми функциями­-селекторами(базовыми функциями доступа к данным).

Car и cdr обеспечивают переход по указателю.

Car и cdr принимают в качестве аргумента точечную пару или список.

Car возвращает голову списка(первый элемент). В пустого списка ­вернет nil.

Cdr возвращает хвост(список, состоящий из всех элементов, кроме первого). Если в списке меньше двух элементов, то функция возвращает nil.

1. **Отличия реализации cons и list**

Функции list и cons являются функциями-­конструкторами.

List - произвольная функция.

Cons - базисная функция.

Cons создает списковую ячейку и устанавливает два указателя на принимаемые два аргумента.

List создает столько списковых ячеек сколько нужно в отличии от cons. List принимает переменное число аргументов и возвращает список, элементами которого являются аргументы функции.

1. **Как синтаксически представляется программа на Lisp**

В Lisp формы представления программы и обрабатываемых ею данных одинаковы (в виде S-выражений).

Поэтому программы могут обрабатывать и преобразовывать другие программы или сами себя.

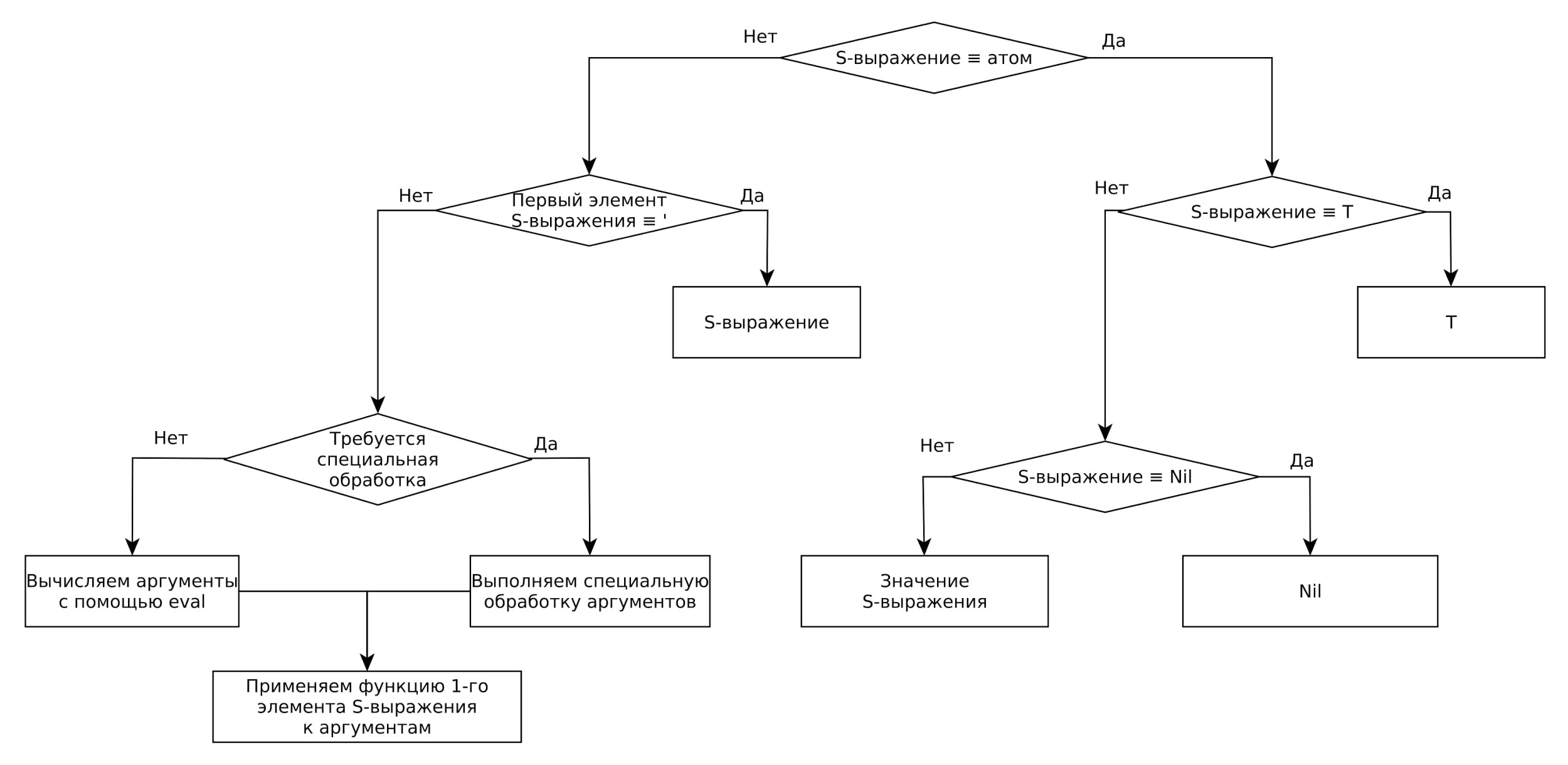
1. **Как трактуются элементы списка?**

Если не стоит блокировка вычисления(quote, ‘), то первый элемент трактуется как имя функции, остальные — как аргументы. Если блокировка стоит, то элементы трактуются как обычные элементы списка.

1. **Порядок реализации программы**

Программы работает в цикле:

* ожидает ввода S-выражения;
* передает введенное S-выражение функции eval;
* выводит полученный результат.



1. **Способы определения функции**

* новые функции можно определить с помощью оператора defun. Он обычно принимает: имя, список параметров и тело функции.

**(defun name (arg1, arg2) (форма))**

форма - s-выражение по которому работает функция

**(defun sum (x y) (+ x y))**

* lambda нотация функции (способ определения функции без имени)

**(lambda (<список аргументов>) (<форма>))**

**(lambda (x) (+ (\* 2 x) (\* 3 x)))**

Позволяет определить новую функцию, у которой не будет имени. Список аргументов (разделяются пробелом) блокировать нельзя.

* вызывается с помощью специальных функций apply и funcall

Apply требует функцию и список ее аргументов и возвращает результат вызова этой функции с заданными аргументами.

**(apply #’ average ‘(5 7)) -> 6.0**

**(apply #’ (lambda (x y) (/ (+ x y) 2.0)) ‘(5 7))**

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

**‘** - блокировка вычислений

**#’** - функциональная блокировка (без пробела - один символьный атом) (стоит перед функцией)

**#’** - сокращение для function. Давая функции имя с помощью function, мы получаем объект.

**> (function +)**

**#<compiled-function +>**

Функция funcall делает тоже самое, что и apply, но не требует, чтобы аргументы были упакованы в список:

**> (funcall #’+ 1 2 3) 6**

1. **Работа and, or, if, cond**

* cond

Сначала просматриваются все предикаты в порядке следования, и если хоть один из них истинный, то cond возвращает результат, связанный с этим предикатом. Если ни один предикат не был истинным, вернется Nil.

**(cond ((условие1) результат1)**

**((условие2) результат2)**

**((условиеN) результатN))**

* if

Сначала выполняется форма test. Если результат не равен nil, тогда выбирается форма true\_body, иначе выбирается форма false\_body.

**(if test true\_body false\_body)**

* and

Логическая функция AND берет один или несколько аргументов. Она выполняет эти аргументы слева направо. Если она встречает аргумент, значение которого NIL, она возвращает NIL, не продолжая вычисления остальных. Если NIL аргументов не встретилось, то возвращается значение последнего аргумента.

**(and arg1 arg2 … argn)**

* or

Логическая функция OR берет один или несколько аргументов. Она выполняет эти аргументы слева направо и возвращает значение первого аргумента, который не NIL.Если все аргументы OR имеют значение NIL, то OR возвращает NIL.

**(or arg1 arg2 … argn)**

1. **Cтруктуроразрушающие и неразрушающие структуру функции**
   * **структуроразрушающие функции** - **функций, которые изменяют содержимое указателей, вместо того чтобы создавать новые списочные ячейки**

nreverse - инвертирует список, не создает копию

nconc - соединяет списки и возвращает результат. При этом аргументы изменяются, а не копируются(теряется возможность работать со списками отдельно)

* + **не разрушающие структуру функции создают новые списочные ячейки(работают медленнее тк копируют)**

append - создает копии всех аргументов кроме последнего. На копирование уходит время.

reverse - инвертирует список(работает с копией)

1. **Отличия в работе cons, list, append**

**Cons** - базисная функция. Cons создает списковую ячейку и устанавливает два указателя на принимаемые два аргумента.

**List** - произвольная функция.List создает столько списковых ячеек сколько нужно в отличии от cons. List принимает переменное число аргументов и возвращает список, элементами которого являются аргументы функции.

**Append** - создает копии всех списочных ячеек для каждого элемента во всех аргументов, исключая последний аргумент, в то время как cons создает только одну списочную ячейку.

Если складывают два списка в 1000 и 1 элемент, то будет создано 1000 копий списочных ячеек, вместо того чтобы исправить один указатель.

1. **Варианты и методы модификации элементов списка**

Common Lisp определяет несколько операций для применения какой-либо функции к каждому элементу списка. Чаще всего для этого используется mapcar, которая вызывает заданную функцию поэлементно для одного или нескольких список и возвращает список результатов:

**(mapcar \#'(lambda (x) (+ x 10)) '(1 2 3)**

**(11 12 13)**

Похожим образом действует maplist, однако применяет функцию последовательно не к car, а к cdr списка, начиная со всего списка целиком.

**maplist \#'(lambda (x) x) '(a b c)**

**((A B C) (B C) (C))**

Среди других функций можно отметить mapc, которая похожа на mapcar, однако она не строит список из вычисленных значений, поэтому может использоваться лишь ради побочных эффектов. Также она возвращает свой последний аргумент.

**(mapc \#'(lambda (x y) (format t "\ ~a ~a "\ x y)) '(hip flip slip) '(hop flop slop))**

**HIP HOP FLIP FLOP SLIP SLOP**

**(HIP FLIP SLIP)**

Функция mapcan похожа на mapcar, но соединяет в один список возвращаемые значения(которые должны быть списками) с помощью nconc:

**(mapcan \#'list '(a b c) '(1 2 3 4))**

**(A 1 B 2 C 3)**

1. **Порядок работы и варианты использования функционалов.**

Аргумент функции, который является функцией называется функциональным, а функция, имеющая хотя бы один функциональный аргумент, называется функционалом.Среди всех мыслимых функционалов можно выделить один специфический вид - функционал, который применяет функциональный аргумент к остальным аргументам. Такие функционалы обычно называют применяющими функционалами.

Еще одним классом функционалов является класс отображающих функционалов. Такие функционалы применяют функциональный аргумент к элементам списка, в результате чего строится новый список. Отсюда и название: исходный список отображается на результирующий.

Все стандартные функции работают только по верхнему уровню списка/

Функции списков:

1. (append list1 list2)

2. (nconc list1 list2)

3. (reverse list1) (nreverse list1)

4. (last lst)

5. (nth N lst)

6. (nthcdr N lst)

nth возвращает значение головы n-й ячейки (нумерация с 0).

nthcdr возвращает значение хвоста n-1-й ячейки.

7. (length lst)

8. (remove el lst) // delete, для нахождения элемента используется eql

9. (rplaca lst el) (rplacs lst el)

10. (subst new old lst) (nsubst new old lst)

Функции множества:

1. (member el lst) // до первого вхождения

2. (union lst1 llst2) // не разрушает структуру

3. (intersection lst1 lst2) // не ращрушает

4. (set=difference lst1 lst2)

5. (subsetp lst1 lst2)

Функции для ассоциативных таблиц:

1. (assoc key table) (key table)

2….

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

# **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.

**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