

Базис Lisp

Базис – это минимальный набор инструментов языка и структур данных, который позволяет решить любые задачи.

Базис Lisp :

- атомы (представляются в памяти пятью указателями – name, value, function, property, package) и структуры (представляющиеся бинарными узлами);
- базовые (несколько) функций, функционалов и форм: встроенные – примитивные функции (atom, eq, cons, car, cdr); формы (quote, cond, lambda, eval); функционалы (apply, funcall).

Атомы:

- символы (идентификаторы) – синтаксически – набор литер (букв и цифр), начинающихся с буквы;
- специальные символы – T, Nil (используются для обозначения логических констант);
- самоопределимые атомы – натуральные числа, дробные числа, вещественные числа, строки – последовательность символов, заключенных в двойные апострофы (например, “abc”);

Более сложные данные – списки и точечные пары (структуры), которые строятся с помощью унифицированных структур – блоков памяти – бинарных узлов.

Определения:

Точечная пара ::= (<атом> . <атом>) | (<атом> . <точечная пара>) | (<точечная пара> . <атом>) | (<точечная пара> . <точечная пара>);

Список ::= <пустой список> | <непустой список>, где

<пустой список> ::= () | Nil,

<непустой список> ::= (<первый элемент> . <хвост>),

<первый элемент> ::= <S-выражение>,

S-выражение ::= <атом> | <точечная пара>,

<хвост> ::= <список>.

Функцией называется правило, по которому каждому значению одного или нескольких аргументов ставится в соответствие конкретное значение результата.

Функционалом, или функцией высшего порядка называется функция, аргументом или результатом которой является другая функция.

Форма – функция, которая особым образом обрабатывает свои аргументы, т. е. требует специальной обработки. Переменное число аргументов или они обрабатываются/не обрабатываются по-разному.

Синтаксически:

любая структура (точечная пара или список) заключается в круглые скобки: (A . B) – точечная пара, (A) – список из одного элемента. Пустой список изображается как Nil или ();

непустой список по определению может быть изображен: (A . (B . (C . (D . ())))), допустимо изображение списка последовательностью атомов, разделенных пробелами – (A B C D).

Элементы списка могут быть списками (любой список заключается в круглые скобки), например – (A (B C) (D C)). Таким образом, синтаксически наличие скобок является признаком структуры – списка или точечной пары.

Любая непустая структура Lisp в памяти представляется списковой ячейкой, хранящей два указателя: на голову (первый элемент) и хвост – всё остальное. Точечная пара в памяти представляется бинарным узлом.

Отличительные особенности Lisp: только символьная обработка; все можно представить в виде функций.

Вся информация (данные и программы) в Lisp представляется в виде символьных выражений – S-выражений.

По определению: S-выражение ::= <атом> | <точечная пара>.

В зависимости от контекста одни и те же объекты могут играть роль переменных или констант, причем значения и того, и другого могут быть произвольной сложности. Если объект играет роль константы, то для объявления константы достаточно заблокировать его вычисление, то есть как бы взять его в кавычки, отмечающие буквально используемые фразы, не требующие обработки.

Апостроф – сокращённое обозначение функции quote.

quote - блокирует вычисление своего аргумента. В качестве своего значения выдаёт сам аргумент, не вычисляя его. Перед константами - числами и атомами T, Nil можно не ставить апостроф.

Синтаксическая форма и хранение программы в памяти

Программа на Lisp представляет собой вызов функции на верхнем уровне. Все операции над данными оформляются и записываются как функции, которые имеют значение, даже если их основное предназначение – осуществление некоторого побочного эффекта. Программа является ничем иным, как набором запрограммированных функций.

Синтаксически программа оформляется в виде S-выражения (обычно – списка – частного случая точечной пары), которое очень часто может быть структурированным. Наличие скобок является признаком структуры.

Атомы представляются **в памяти** пятью указателями (name, value, function, property, package), а любая непустая структура – списковой ячейкой (бинарным узлом), хранящей два указателя: на голову (первый элемент) и хвост – все остальное.

Трактовка элементов списка.

По определению списка, приведенному выше: если список непустой, то он представляет из себя точечную пару из <первого элемента> и <хвоста>, где <первый элемент> – это <S-выражение>, а <хвост> – это <список>.

Список можно вычислить, если он представляет собой обращение к функции, или функциональный вызов: $(f\ e1\ e2\ \dots\ en)$, где f – символьный атом, имя вызываемой функции; $e1, e2, \dots, en$ – аргументы этой функции; n – число аргументов функции.

В случае $n=0$ имеем вызов функции без аргументов: (f) . Обычно $e1, e2, \dots, en$ являются вычислимыми выражениями и вычисляются последовательно слева направо.

Таким образом, если в процессе работы лисп-интерпретатора требуется вычислить некоторый список, то первым элементом этого списка должно быть имя функции. Если это не так, лисп-интерпретатор сообщает об ошибке и прерывает вычисление текущего выражения программы.

Порядок реализации программы.

Типичная лисп-программа включает:

- определения новых функций на базе встроенных функций и других функций, определённых в этой программе;
- вызовы этих новых функций для конкретных значений их аргументов.

Как отмечалось выше, программа на Lisp представляет собой вызов функции на верхнем уровне и синтаксически оформляется в виде S-выражения. Вычисление программы реализует лисп-интерпретатор, который считывает очередную входящую в программу форму, вычисляет её (анализирует функцией `eval`) и выводит полученный результат (S-выражение).

`Eval` выполняет двойное вычисление своего аргумента. Эта функция является обычной, и первое вычисление аргумента выполняет так же, как и любая обычная функция. Полученное при этом выражение вычисляется ещё раз. Такое двойное вычисление может понадобиться либо для снятия блокировки вычислений (установленной функцией `quote`), либо же для вычисления сформированного в ходе первого вычисления нового функционального вызова.

Вызов (`eval S-выражение`):

Классификация функций

Функции – лишь логическая интерпретация, а синтаксически все одинаково.

Функции делятся на базисные или небазисные, и по способу реализации (чистые математические и тд)

Формальные параметры – символьные атомы. Когда задаем фактические параметры, должна быть выделена память. Лексическая переменная – символьный атом. Распределение памяти автоматически в нужный момент.

1. Чистые математические функции (имеют фиксированное количество аргументов, сначала выясняются все аргументы, а только потом к ним применяется функция);
2. Рекурсивные функции (основной способ выполнения повторных вычислений);
3. Специальные функции, или формы (могут принимать произвольное количество аргументов, или аргументы могут обрабатываться по-разному);
4. Псевдофункции (создают «эффект», например, вывод на экран);
5. Функции с вариантами значений, из которых выбирается одно;
6. Функции высших порядков, или функционалы – функции, аргументом или результатом которых является другая функция (используются для построения синтаксически управляемых программ);

Классификация базисных функций и функций ядра.

1. Селекторы: `car` и `cdr` (будут подробнее рассмотрены ниже).
2. Конструкторы: `cons` и `list` (будут подробнее рассмотрены ниже).
3. Предикаты – «логические» функции, позволяющие определить структуру элемента:
 - `atom` возвращает `T`, если значением её единственного аргумента является атом, иначе – `NIL`;
 - `null` возвращает `T`, если значение его аргумента – `NIL` (пустой список), иначе – `NIL`;
 - `listp` возвращает `T`, если значением её аргумента является список, иначе – `NIL`;
 - `conspr` возвращает `T`, если значением её аргумента является структура, представленная в виде списковой ячейки, иначе – `NIL`.
4. Функции сравнения (принимают два аргумента, перечислены по мере роста «тщательности» проверки):
 - `eq` корректно сравнивает два символьных атома. Так как атомы не дублируются для данного сеанса работы, то фактически сравниваются соответствующие указатели. Возвращает `T`, когда:
1) значением одного из аргументов является атом, и одновременно
2) значения аргументов равны (идентичны). В ином случае значением функции `eq` является `NIL`. (`eq 'ab 'Ab`) => `T`, но (`eq 1 2`) => `NIL`.

- `eq1` корректно сравнивает атомы и числа одинакового типа (синтетической формы записи). Например, `(eq1 1 1)` вернет `T`, а `(eq1 1 1.0)` – `Nil`, так как целое значение `1` и значение с плавающей точкой `1.0` являются представителями различных классов;
- `=` корректно сравнивает только числа, причем числа могут быть разных типов. Например, `(= 1 1)`, и `(= 1 1.0)` вернет `T`;
- `equal` работает идентично `eq1`, но в дополнение умеет корректно сравнивать списки (считая списки эквивалентными, если они рекурсивно, согласно тому же `equal`, имеют одинаковую структуру и содержимое; считая строки эквивалентными, если они содержат одинаковые знаки);
- `equalp` корректно сравнивает любые `S`-выражения.

Функционалы

Функционалы: 2 группы - Применяющие (`apply`, `funcall`) и отображающие (`mapcar`, `maplist`)

Функция – всегда первый аргумент функционала

Применяющие: (`apply #'fun lst`) – `fun`-имя или `lambda`-выражение, `lst` – список (фиксированное число), (`funcall #'fun arg1, ... argn`) – количество аргументов определяется `fun`.

Фиксирует окружение функции в момент, когда функция начала работать (может, в теле есть глобальные атомы)

Отображающие - функционалы, которые позволяют реализовывать многократные или повторные вычисление (циклы).

(`mapcar #'fun lst`) – `fun` – имя или `lambda`-выражение, к каждому элементу списка по верхнему уровню, на выходе – список из результатов. функция должна уметь обрабатывать как атомы, так и списки получается

(`maplist #'fun lst`) – целиком к списку, к хвосту, к хвосту хвоста ..., функция должна работать со списками

Функция должна быть одноаргументной.

В результирующий список все объединяется функцией `list`.

`Fun` должна быть одноаргументной. В предыдущих примерах. Чтобы использовать многоаргументные:

(`mapcar #'fun lst1 lst2 .. lstk`) Выбирает из каждого списка `car`-элементы и применяет функцию, затем – вторые (головы от хвостов). Проход по верхнему уровню. Закончится, когда закончатся элементы самого короткого списка.

Аналогично `maplist`

Существуют аналог `maplist` – `mapcon`, использующие структуроразрушающий `pnconc` вместо `list`. Работает быстрее, НО результаты применения функций должны быть списками, для `pnconc`. `Mapcar` – `mapcar`

- `Find-if` (функция или предикат (`find-if #'predicat lst`) - проходит по верхнему уровню списка и возвращает первый элемент, удовлетворяющий

данному предикату (find-if #'odd. '(2 4 7 5)) -> 7 (find-if-not #'predicat lst) – не удовлетворяющий • Remove-if (remove-if #'predicat lst) (remove-if-not #'predicat lst)

- Reduce (reduce #'fun lst) Fun – как минимум 2-аргументная. Применяет функцию каскадным образом к элементам lst – 1 и 2, к результату и 3. • (every #'predicat lst) – Т если все элементы удовлетворяют предикату
- (some #'predicat lst) – Т если некоторые

Примеры на использование функционалов

- Множество из списка (defun consists-of (lst) (if (member (car lst) (cdr lst)) 1 0)) (defun all-last-element (lst) (if (eql (consists-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) – множество, но порядок «неожиданный»

1. Декартово произведение (defun decart (listx listy) (mapcan #'(lambda (x) (mapcar #'(lambda (y) (list x y)) listy)) listx))

#' нужно использовать во вложенной функции, чтобы зафиксировать значение x. На более высоком уровне. В ЛАБАХ ВСЕГДА ПИСАТЬ

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

Структуроразрушающие и не разрушающие структуры списка функции

Функции, работающие со списками, делятся на:

- функции, не разрушающие структуру списка (сохраняется возможность работать с исходным списком);
- функции, разрушающие структуру списка (не сохраняется возможность работы с исходным списком, зато функция выполняется быстрее по сравнению со своим не разрушающим аналогом, так как не создаются копии cons-ячеек).

Из-за такого разделения существует много дублирующих функций: функциям, не разрушающим структуру списка (reverse, substitute, ...) соответствуют структуроразрушающие функции, которые, как правило, начинаются с буквы «n», как признак того, что не создаются копии (nreverse, nsubstitute, ...). cons является структуроразрушающим аналогом append, delete – структуроразрушающим аналогом remove.

Способы создания функций

Определение функций пользователя в Lisp-е возможно двумя способами.

- Базисный способ определения функции - использование λ -выражения (λ -нотации). Так создаются функции без имени.

λ -выражение: (lambda λ -список форма), где λ -список – это формальные параметры функции (список аргументов), а форма – это тело функции.

Вызов такой функции осуществляется следующим способом: (λ -выражение последовательность_форм), где последовательность_форм – это фактические параметры.

Вычисление функций без имени может быть также выполнено с использованием функционала apply: (apply λ -выражение последовательность_форм), где последовательность_форм – это список фактических параметров; или с использованием функционала funcall: (funcall λ -выражение последовательность_форм), где последовательность_форм – это фактические параметры.

Функционал apply является обычной функцией с двумя вычисляемыми аргументами, обращение к ней имеет вид: (apply F L), где F – функциональный аргумент и L – список, рассматриваемый как список фактических параметров для F. Значение функционала – результат применения F к этим фактическим параметрам.

Функционал funcall – особая функция с вычисляемыми аргументами, обращение к ней: (funcall F e1 ... en), n 0. Её действие аналогично apply, отличие состоит в том, что аргументы применяемой функции F задаются не списком, а по отдельности.

funcall используется тогда, когда во время написания кода количество аргументов известно, apply – когда неизвестно.

- Другой способ определения функции – использование макро-определения defun:

(defun имя_функции λ -выражение),

или в облегченной форме:

(defun имя_функции (x_1, x_2, \dots, x_k) форма), где (x_1, x_2, \dots, x_k) – это список аргументов.

В качестве имени функции выступает символьный атом. Вызов именованной функции осуществляется следующим образом: (имя_функции последовательность_форм), где последовательность_форм – это фактические параметры. Также для ее вызова можно воспользоваться рассмотренными выше функционалами funcall (например, (foo 1 2 3) === (funcall #'foo 1 2 3)) и apply (например, (apply #'plot plot-data), где plot-data – список, хранящий аргументы).

λ -определение более эффективно, особенно при повторных вычислениях.

Параметры функции, переданные при вызове, будут связаны с переменными в списке параметров из объявления функции. Еще один способ связывания формальных параметров с фактическими – использование функции let:

(let ((x1 p1) (x2 p2) ... (xk pk)) e),
где x_i – формальные параметры, p_i – фактические параметры (могут быть формами), e – форма (что делать).

Функции Car и Cdr

Функции `car` и `cdr` переходят по соответствующему указателю аргумента (бинарного узла).

Функция `car` от одного аргумента возвращает первый элемент списка, являющегося значением её аргумента.

Функция `cdr` возвращает хвост списка, являющегося значением её единственного аргумента (хвостом, или остатком списка является список без своего первого элемента).

Современные диалекты Лиспа обычно допускают для функций `car` и `cdr` мнемоничные синонимичные названия: `first` и `rest` соответственно.

Eval, квотирование

`Eval` выполняет двойное вычисление своего аргумента. Эта функция является обычной, и первое вычисление аргумента выполняет так же, как и любая обычная функция. Полученное при этом выражение вычисляется ещё раз. Такое двойное вычисление может понадобиться либо для снятия блокировки вычислений (установленной функцией `quote`), либо же для вычисления сформированного в ходе первого вычисления нового функционального вызова.

Вызов (`eval S-выражение`):

[scale=0.5]img/eval

Квотирование объекта – это применение к нему функции `quote`, которая в качестве своего значения выдаёт сам аргумент, не вычисляя его. По сути, эта функция блокирует вычисление своего аргумента. Необходимость в этом нередко возникает при использовании обычных встроенных функций, чтобы задать аргументы в явном виде и избежать их вычисления. Константы–числа и атомы `T`, `NIL` при их использовании в качестве аргументов обычных функций можно не квотировать, поскольку значением любой константы является она сама. Функция `quote` используется часто, поэтому допускается упрощённый способ обращения к ней с помощью апострофа, маркирующего котируемое выражение.

В диалекте `Common Lisp` для замыкания функционального аргумента встроена специальная форма (`function F`), где F – определяющее выражение функции. Эту форму часто называют функциональной блокировкой, поскольку она аналогична по действию функции `quote`, но не просто котирует аргумент, а как бы замыкает значения используемых в функциональном аргументе F свободных переменных, фиксируя их значения из контекста

его определения. Функциональную блокировку можно записывать короче, с помощью двух знаков `#'` (получить функцию с данным именем").

Cond, if, and, or, not

cond

Общий вид условного выражения:

$(cond (p_1 e_{11} e_{12} \dots e_{1m_1}) (p_2 e_{21} e_{22} \dots e_{2m_2}) \dots (p_n e_{n1} e_{n2} \dots e_{nm_n})), m_i; 0, n1$

Вычисление условного выражения общего вида выполняется по следующим правилам:

1. последовательно вычисляются условия p_1, p_2, \dots, p_n ветвей выражения до тех пор, пока не встретится выражение p_i , значение которого отлично от NIL;
2. последовательно вычисляются выражения-формы $e_{i1} e_{i2} \dots e_{im_i}$ соответствующей ветви, и значение последнего выражения e_{im_i} возвращается в качестве значения функции cond;
3. если все условия p_i имеют значение NIL, то значением условного выражения становится NIL.

Ветвь условного выражения может иметь вид (p_i) , когда $m_i = 0$. Тогда если значение $p_i \neq \text{NIL}$, значением условного выражения cond становится значение p_i .

В случае, когда $p_i \neq \text{NIL}$ и $m_i \geq 2$, то есть ветвь cond содержит более одного выражения e_i , эти выражения вычисляются последовательно, и результатом cond служит значение последнего из них e_{im_i} . Таким образом, в дальнейших вычислениях может быть использовано только значение последнего выражения, и при строго функциональном программировании случай $m_i \geq 2$ обычно не возникает, т.к. значения предшествующих e_{im_i} выражений пропадают.

if

Макрофункция (If C E1 E2), встроенная в MuLisp и Common Lisp, вычисляет значение выражения E1, если значение выражения C отлично от NIL, в ином случае она вычисляет значение E2:

$(defmacro \text{If} (C E1 E2) (list 'cond (list C E1) (list T E2)))$

Этот макрос строит и вычисляет условное выражение cond, в котором в качестве условия первой ветви берётся выражение C (первый аргумент If), а выражения E1 и E2 (второй и третий аргумент If) размещаются соответственно на первой и второй ветви cond.

and/or/not

К логическим функциям-предикатам относят логическое отрицание not, конъюнкцию and и дизъюнкцию or. Первая из этих функций является обычной, а другие две — особыми, поскольку допускают произвольное количество аргументов, которые не всегда вычисляются.

Логическое отрицание `not` вырабатывает соответственно: $(\text{not NIL}) \Rightarrow T$ и $(\text{not } T) \Rightarrow \text{NIL}$, и может быть определено функцией $(\text{defun not } (x) (\text{eq } x \text{ NIL}))$.

Вызов функции `and`, реализующей конъюнкцию, имеет вид $(\text{and } e1\ e2\ \dots\ en)$, $n \geq 0$.

При вычислении этого функционального обращения последовательно слева направо вычисляются аргументы функции e_i — до тех пор, пока не встретится значение, равное `NIL`. В этом случае вычисление прерывается и значение функции равно `NIL`. Если же были вычислены все значения e_i и оказалось, что все они отличны от `NIL`, то результирующим значением функции `and` будет значение последнего выражения e_n .

Вызов функции-дизъюнкции имеет вид $(\text{or } e1\ e2\ \dots\ en)$, $n \geq 0$.

При выполнении вызова последовательно вычисляются аргументы e_i (слева направо) — до тех пор, пока не встретится значение e_i , отличное от `NIL`. В этом случае вычисление прерывается и значение функции равно значению этого e_i . Если же вычислены значения всех аргументов e_i , и оказалось, что они равны `NIL`, то результирующее значение функции равно `NIL`.

При $n=0$ значения функций: $(\text{and}) \Rightarrow T$, $(\text{or}) \Rightarrow \text{NIL}$.

Таким образом, значение функции `and` и `or` не обязательно равно `T` или `NIL`, а может быть произвольным атомом или списочным выражением.

Отличие в работе функций `cons`, `list`, `append`, `pncons` и в их результате

`cons` принимает 2 указателя на любые S-выражения и возвращает новую `cons`-ячейку (списковую ячейку), содержащую 2 значения. Если второе значение не `NIL` и не другая `cons`-ячейка, то ячейка печатается как два значения в скобках, разделённые точкой (так называемая точечная пара). Иначе, по сути, эта функция включает значение первого аргумента в начало списка, являющегося значением второго аргумента.

Функция `list`, составляющая список из значений своих аргументов (у которого голова — это первый аргумент, хвост — все остальные аргументы), создает столько списковых ячеек, сколько аргументов ей было передано. Эта функция относится к особым, поскольку у неё может быть произвольное число аргументов, но при этом все аргументы вычисляются.

`append` принимает произвольное количество аргументов-списков и соединяет (сливает) элементы верхнего уровня всех списков в один список. Действие `append` иногда называют конкатенацией списков. В результате должен быть построен новый список.

Например: $(\text{append } (\text{list } 1\ 2) (\text{list } 3\ 4)) \Rightarrow (1\ 2\ 3\ 4)$.

С точки зрения функционального подхода, задача функции `append` — вернуть список $(1\ 2\ 3\ 4)$ не изменяя ни одну из `cons`-ячеек в списках-аргументах $(1\ 2)$ и $(3\ 4)$. `append` на самом деле создаёт только две новые `cons`-ячейки,

чтобы хранить значения 1 и 2, соединяя их вместе и делая ссылку из CDR второй ячейки на первый элемент последнего аргумента - списка (3 4). После этого функция возвращает cons-ячейку содержащую 1. Ни одна из входных cons-ячеек не была изменена, и результатом, как и требовалось, является список (1 2 3 4). Единственная хитрость в том, что результат, возвращаемый функцией append имеет общие cons-ячейки со списком (3 4). Таким образом, если последний переданный список будет модифицирован, то итоговый список будет также изменен.

nconc – это структуроразрушающая версия append. Как и append, nconc возвращает соединение своих аргументов, но строит такой результат следующим образом: для каждого непустого аргумента-списка, nconc устанавливает в cdr его последней cons-ячейки ссылку на первую cons-ячейку следующего непустого аргумента-списка. После этого она возвращает первый список, который теперь является головой результата-соединения.

Итак, отличия: cons является базисной, list, append, nconc – нет; list, append, nconc принимают произвольное количество аргументов (причем аргументами append и nconc могут быть только списки), cons – фиксированное (два); cons создает точечную пару или список (в зависимости от второго аргумента), list, append и nconc – список; cons и list создают новые списковые ячейки (все), append имеет общие списковые ячейки с последним списком, nconc не создает cons-ячеек; cons является структуроразрушающей, а cons, list и append – нет.

Пусть (setf lst1 '(a b)); (setf lst2 '(c d)).

[language=Lisp] (cons lst1 lst2) => ((A B) C D) lst1 => (A B) (list lst1 lst2) => ((A B) (C D)) lst1 => (A B) (append lst1 lst2) => (A B C D) lst1 => (A B) (nconc lst1 lst2) => (A B C D) lst1 => (A B C D)

Про переменные

Лисп работает только на указателях, поэтому все в куче

Формальные параметры – символьные атомы. Когда задаем фактические параметры, должна быть выделена память. Лексическая переменная – символьный атом. Распределение памяти автоматически в нужный момент

Можно сказать, что есть локальные и глобальные атомы (T, Nil).

Установка значения символьному атому – setf,setq от двух аргументов value значение (другой символьный атом, самовычисляемый атом, структура). value устанавливается на значение второго аргумента. Как только написали setf/setq – организовался глобальный атом, то есть на все время сеанса работы. Можно менять значение по указателю, но сам атом уже выделен.

Список свойств – список (динамическая структура) из (имя_свойства значение).

let

(let ((name1 value1) ... (namen valuen)) body)

выделяется память, готовятся значения, а связывание происходит в произвольном порядке. Нельзя из value2 сослаться на value1.

В `let*` - последовательно, более аккуратно, можно сослаться, но она дольше.

Термин среда – при загрузке пакета. Загрузка пакета – загрузка интерфейса (окружение).

Еще функции

`last`

`(nth n lst)`

`(nthcdr n lst)`

remove принимает 2 аргумента и возвращает список, заданный вторым аргументом, из которого удалены все вхождения значения первого аргумента.

substitute принимает 3 аргумента и возвращает список, заданный третьим аргументом, в котором все вхождения значения второго аргумента заменены на значение первого аргумента.

`(remove el lst)` – сравнивает элементы по `car`-указателю очередного элемента

`(member elem, lst)` – возвращает остаток списка, начиная с ячейки, у которой `car` на `elem` указывает.

Как повлиять на работу для поиска списка. Есть механизм ключевых слов. (совокупность ключевых параметров). Могут быть использованы значения по умолчанию `(member '(a b) '(c (a b) d) :test 'equal) => ((a b) d)`

`Assoc/rassoc` - таблицы

`elem` не может быть структурой, так как сравнивается по `eq`.

Еще инфы

Виды списков – одноуровневые и структурированные (элементами являются списки), смешанные и числовые.

Все стандартные функции работают только по верхнему уровню.

Ассоциативная таблица (приближение к БД). – список из точечных пар (ключ – значение), но `lisp` не следит за уникальностью ключей.

Множество – неупорядоченная совокупность неповторяющихся элементов.

`#'` –функциональная блокировка `=function`