Функции работы со множествами:

(member el lst)

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

Продолжение про функционалы

Функция – всегда первый аргумент функционала)

* Применяющие (apply, funcall) и
* отображающие (mapcar, maplist)

!!!!

В результирующий список все объединяется функцией lsit.

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 структуроразрущающий nconc вместо list. Работает быстрее, НО результаты применения функций должны быть списками, для nconc

Mapcan -- 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) – T если все элементы удовлетворяют предикату
* (some #’predicat lst) – T 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)) lsty))

Lstx))

#’ 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))

# Рекурсия

Плохо организованная рекурсия – плохо, хорошо – хорошо и сравнима с итерацией.

Рекурсия – ссылка на описываемый объект при описании объекта.

Надо думать:

* Когда выйти из рекурсии
* Как уйти в рекурсию (как меняются аргументы)
* Как вызвать рекурсивную функцию первый раз

Классификация:

1. Простая рекурсия (рекурсивный вызов встречается в теле функции 1 раз)
2. Рекурсия 1 порядка (несколько раз)
3. Взаимная рекурсия (несколько функций, вызывающие друг друга)

Стремиться к эффективному варианту

Реализация рекурсии с помощью cond.

**Золотое правило** – в качестве первых веток – условия выхода из рекурсии

Пример на факториале.

Если уходим в рекурсию, а собираем результаты только на выходе — это неэффективно. А если сделать предварительные действия на входе. Хвостовая рекурсия

Append всегда делает копии, поэтому в рекурсии ее использовать очень. Плохо

(defun my-member (el lst)

(cond ((nul lst) Nil)

((equal el (car lst)) t)

(t (my-member el (cdr lst)))))

(my-member ‘a ‘(b a c)) -> t

(my-member Nil ())->nil

Если переставить условия выхода, то my-member Nil ())->t

(defun my-reverse (lst)

(cond ((null. Lst) lst)

(t (append (my-reverse (cdr lst)) (cons (car lst Nil)))))

не очень, так как собираем результат на выходе

Перепишем на хвостовую рекурсию

(defun move-to (lst result)

(cond ((null lst) result)

(t (move\_to (cdr lst) (cons (car lst) result)))))

(defun. my-reverse1 (lst)

(move-to lst ())