Работа со списками

Со списками можно работать, как с множествами. Не все элементы списка могут быть обработаны, если функция для работы с множествами. Существует стандартная функция, которая работает со множествами- unit.

Работа с множествами = работа с одноуровневым списком.

Ассоциативные таблицы – представляется в виде списков(точечных пар, которые воспринимаются как ( (key1 value1) ( key2.value2) … ) )

( assoc key Table)

( rassoc value Table)

# Функционалы

Делятся на 2 группы:

1. Применяющие.
   1. (apply #’fun ‘(arg1 … argN )) //блокировка текущего состояния переменных
   2. (funcall #’fun ‘arg1 ‘arg2)
2. Отображающие.  
   Используется особым образом, применяются многократно, позволяют применить функцию к другим аргументам.
   1. (mapcar #’fun list1) – применяет к каждому элементу списка list1, результат множество, объединяет в один общий список.

(mapcar #’fun lst1 lst2) работает так- выбирает car элементов, применяется функция fun, выбираются следующие аргументы и т.д. Работа продолжается до тех пор, пока списки не закончатся, тогда результат возвращается в mapcar

(mapcar #’(lambda(x) (\* x x)) ‘(1 2 3 4)) -> (1 4 9 16)

* 1. (maplist #’fun lst) – fun применяется многократно, ко всему списку, к хвосту, к хвосту хвоста и т.д. Результатов много, объединяются в список, результат maplist будет списком.

(maplist #’reverse ‘(a b c) ) fun применяется многократно, ко всему списку, к хвосту, к хвосту хвоста и т.д. В результате будет такая конструкция ( (c b a) ( c b) (c) )  
Может принимать как1 аргумент, так и результат функции от нескольких аргументов

(maplist #’fun lst1 lst2)

* 1. (find-if #’predicat list)  
       
     (find-if #’odd ‘(2 4 7 9 21) ) -> 7  
     Работает пока не найдет нужный элемент(нечетный)
  2. (remove-if #’predicat lst)
  3. (reduce #’fun lst)- применяет каскадно к списку к списку lst
  4. (reduce #’+ ‘(1 2 3 4) ) -> 10

Примеры использования функционалов:

* (defun decart (lstX lstY)  
   (mapcan #’(lambda (x) (mapcar) #’(lambda(y)))  
   (list x y) ) lsty))  
   lstX))  
  список двухэлементный, применяется многократно в lambda  
  mapcan строит декартово произведение - (decant ‘(a b) ‘(1 2))  
    
  mapcan – mapcar, но не исп дубли к списку  
  mapcon- maplist, но не используются дубли к списку
* (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)

Эффективность - вопрос реализации (с)

Рекурсия:

* Простая - встречается в теле 1 раз
* 1го порядка - встречается несколько раз в теле
* Взаимная - описывается несколько функций, которые рекурсивно вызывают друг друга
* Дополняемая- при обращении рекурсивной функции исп доп функция, но не в качестве аргумента вызова, а вне его

(defun fn (x)  
 (cond (end-test end-value)   
 (t (add-function add\_value  
 (fn (changed\_x))))))

(defun my\_length (lst)  
 (cond ((null est) 0)  
 (t (+ (my\_length (cdr lst))))))

(defun fn (x)  
 (cond (end\_test end\_value)  
 (t (cons new\_el  
 (fn (changed\_x))))))

Рекурсия часто организуется функцией cond

(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)))))

Для преобразования не хвостой рекурсии в хвостовую, рекомендуется использовать дополнительный параметр, в котором постепенно будет формироваться результат работы рекурсивной функции. Таким образом, в описании функции возникает элементов больше, чем требуется пользователю. Очень часто первый вызов рекурсивной функции выполняют с начальным значением параметра для предназначенного результата, равным либо пустому списку, либо нулю(смотря с чем вы работаете)