

Функциональное и логическое программирование

Лекция 3

1.9. Внутреннее представление s-выражений

Атом → информационная ячейка (ячейка памяти).

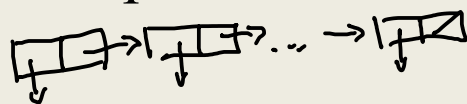
Атом заменяется во внутреннем представлении на адрес информационной ячейки.

Через информационную ячейку можно получить доступ к списку свойств атома, среди которых содержится как внешнее представление, так и указатель на значение.

Оперативная память логически разбивается на списочные ячейки, состоящие из двух полей с указателями. Каждый указатель может ссылаться на другую списочную ячейку или объект Лиспа:

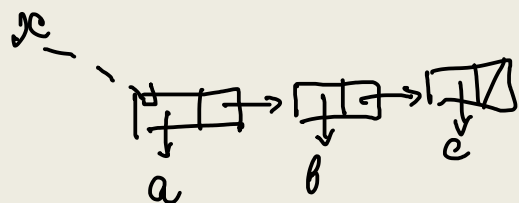


Список – последовательность списочных ячеек, связанных через указатели в правой части.



Пример 1:

Рассмотрим функцию (**SETQ** x '(a b c)) \rightarrow (a b c)



Пунктиром выделен побочный эффект.

Исходя из графического представления становится понятной работа функций **CAR**, **CDR**, **CONS**.

Функция **CAR** возвращает значение левой списочной ячейки.

Функция **CDR** возвращает значение правой списочной ячейки.

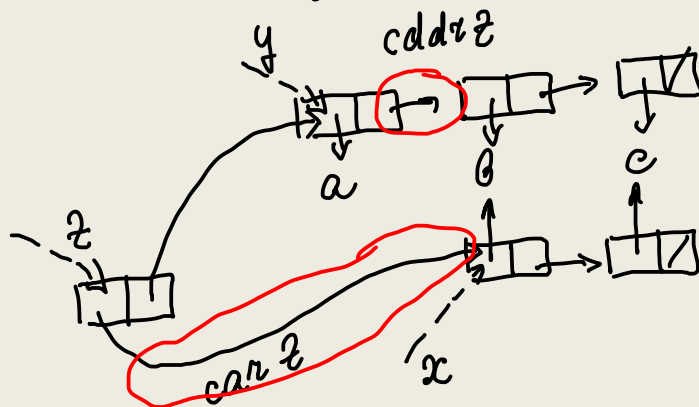
Функция **CONS** создает новую списочную ячейку, содержимое левого поля которого – это указатель на первый аргумент функции, а содержимое правого поля – это указатель на второй аргумент функции.

Пример 2:

(SETQ y '(a b c)) \rightarrow (a b c)

(SETQ x '(b c)) \rightarrow (b c)

(SETQ z (CONS x y)) \rightarrow ((b c) a b c)



Идентичные атомы содержатся в структуре один раз.
Логически идентичные списки могут быть представлены различными списочными ячейками.

(CAR z) \rightarrow (b c)

(CDDR z) \rightarrow (b c)

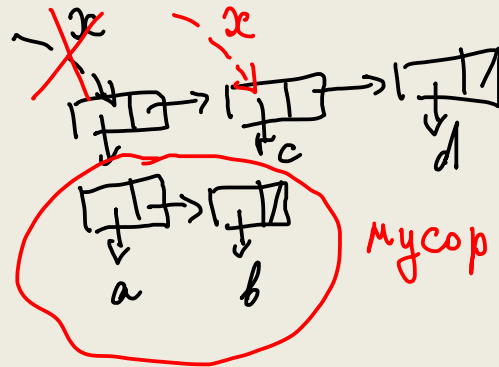
(equal (car z) (caddr z)) \rightarrow t

(eq (car z) (caddr z)) \rightarrow nil

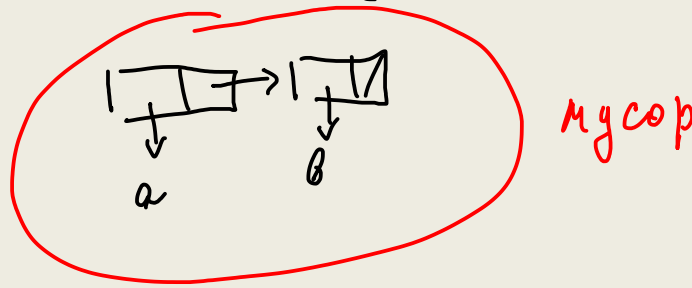
В результате вычислений в памяти могут возникнуть структуры, на которые нельзя сослаться. Такие структуры называются *мусором*.

Примеры образования «мусора»:

1. $(\text{SETQ } x \text{ '}((a \ b) \ c \ d)) \rightarrow ((a \ b) \ c \ d)$
 $(\text{SETQ } x (\text{CDR } x)) \rightarrow (c \ d)$



2. (CONS 'a (LIST 'b)) \rightarrow (a b)



Для повторного использования ставшей мусором памяти в Лиспе предусмотрен специальный сборщик мусора, который автоматически запускается, когда в памяти остается мало свободного места.

Все рассмотренные до сих пор функции манипулировали выражениями, не меняя существующие структуры.

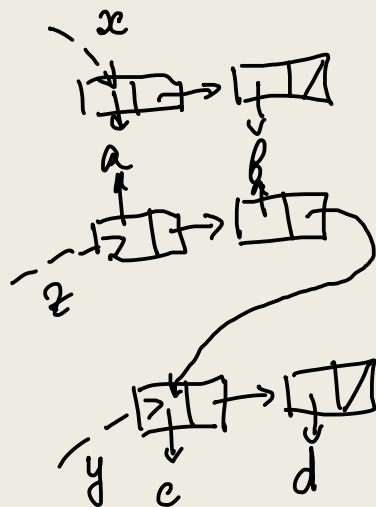
В Лиспе есть специальные функции, которые изменяют внутреннюю структуру списков - структуроразрушающие функции.

Пример 3 (работа функции **APPEND**):

(**SETQ** x '(a b)) → (a b)

(**SETQ** y '(c d)) → (c d)

(**SETQ** z (**APPEND** x y)) → (a b c d)



Очевидно, что если первый аргумент функции **APPEND** является списком из 1000 элементов, а второй – списком из одного элемента, то будет создано 1000 новых ячеек, хотя нужно добавить всего лишь один элемент к 1000 имеющимся.

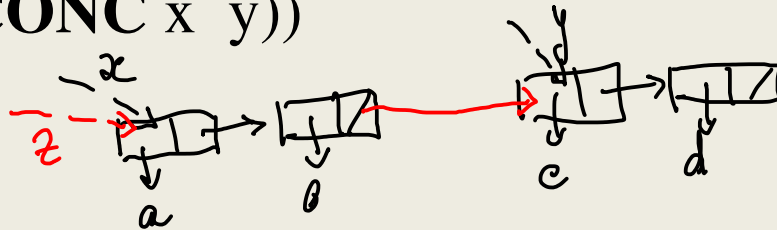
Если нам не важно, что значение переменной *x* может измениться, то можно использовать соединение списков с помощью структуроразрушающей функции **NCONC**:

(NCONC sp₁ sp₂ ... sp_n)

(SETQ x '(a b)) → (a b)

(SETQ y '(c d)) → (c d)

(SETQ z (NCONC x y))



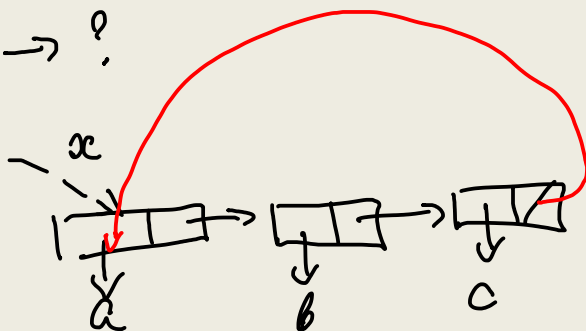
z → (a b c d)
x → (a b c d)

NCONC может создавать циклические структуры

Пример 4:

(SETQ x '(a b c)) \rightarrow (a b c)

(NCONC x x) \rightarrow ?



(a b c a b c)

Еще 2 функции, изменяющие структуру своих аргументов.

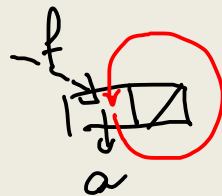
RPLACA (список s-выражение)

Заменяет указатель на голову списка на значение s-выражения.

Пример 5:

(SETQ f '(a)) \rightarrow (a)

(RPLACA f f)



(((...

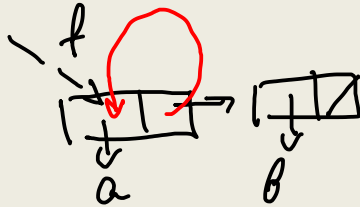
RPLACD (список s-выражение)

Заменяет указатель на хвост списка на значение s-выражения.

Пример 6:

(SETQ f '(a b)) → (a b)

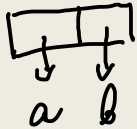
(RPLACD f f) →



(a a a ...)

1.10 Точечная пара

$(\text{CONS } 'a \ 'b) \rightarrow (a . b)$



Любой список можно представить в точечной нотации. Преобразования можно осуществить следующим образом: каждый пробел заменяется точкой, за которой ставится открывающаяся скобка. Соответствующая закрывающаяся скобка ставится непосредственно перед ближайшей справа от этого пробела закрывающейся скобкой, не имеющей парной открывающей скобки также справа от пробела. После каждого последнего элемента списка добавляется `.nil`.

Переход к точечной нотации:

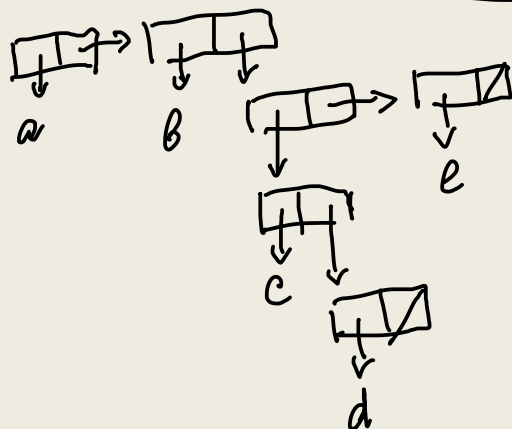
$(a_1 \ a_2 \ \dots a_n) \Leftrightarrow (a_1 . (a_2 . \dots (a_n . \text{nil}) \dots))$

Пример 1:

(a b (c d) e)

Эквивалентное представление с помощью точечных пар:

$(a . (\underbrace{b . ((c . (d . \text{nil})) . (e . \text{nil}))}))$



Записанное в точечной нотации выражение можно частично или полностью привести к списочной нотации.

Переход к списочной записи осуществляется по следующему правилу: если точка стоит перед открывающейся скобкой, то она заменяется пробелом и одновременно убирается соответствующая закрывающаяся скобка. Это же правило позволяет избавиться и от лишних `nil`, если помнить, что `nil` эквивалентен `()`.

Пример 2:

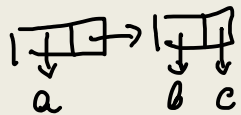
$(a \rightarrow ((b \rightarrow \text{nil}) \rightarrow (c \rightarrow \text{nil})))$

Эквивалентное представление с помощью списка:

$$(a \rightarrow (b \rightarrow c \rightarrow \text{nil})) \Leftrightarrow (a \rightarrow (b \rightarrow c))$$

$(a \rightarrow (b \rightarrow c))$

Эквивалентное представление с помощью списка:

$$(a \rightarrow b \rightarrow c)$$


1.11 Функционалы

В Лиспе функции могут выступать в качестве аргументов (аргументом функции может быть определяющее функцию лямбда-выражение или имя другой функции). Такой аргумент называется *функциональным*, а функция, имеющая функциональный аргумент, называется *функционалом*.

1.11.1 Аппликативные (применяющие) функционалы

Применяющим функционалом называется функционал, который применяет функциональный аргумент к остальным параметрам.

(APPLY fn sp)

Вычисляет значение функционального аргумента (функции от n переменных) для фактических параметров, которые являются элементами списка.

Пример 1:

Написать функциональный предикат **ALL**, который возвращает **t** в том и только в том случае, если функциональный аргумент истинен для каждого элемента списка.

```
(defun all (p l)
  (cond
    ((null l) t)
    ((apply p (list (car l) (all p (cdr l))))
     (t nil))
  )
)
```

(all 'numberp '(1 2 a c)) \rightarrow nil

(all (lambda (x) (> x 100)) '(101 110 102)) \rightarrow t

(**FUNCALL** fn v_1 v_2 ... v_n)

Работает аналогично **APPLY**, но аргументы функционального аргумента (функции от n переменных) задаются не списком, а как аргументы **FUNCALL**, начиная со второго.

Пример 2:

Написать функцию сортировки списка методом вставки в виде функционала **SORT1**, у которого функциональный аргумент будет задавать порядок сортировки.

```

(defun sort1 (p l)
  (cond
    ((null l) l)
    (t (add p (car l) (sort1 p (cdr l)))))
  )

```

```

(defun add (p x l)
  (cond
    ((null l) (list x))
    ((funcall p x (car l)) (cons x l))
    (t (cons (car l) (add p x (cdr l)))))
  )

```

```

(sort1 '> '(3 1 5 0)) → (5 3 1 0)
(sort1 '< '(3 1 5 0)) → (0 1 3 5)
(sort1 'string< '(b a d e)) → (a b d e)

```

1.11.2 Отображающие функционалы или MAP-функции

Отображающие функционалы с помощью функционального аргумента преобразуют список в новый список или порождают побочный эффект, связанный с этим списком. Такие функционалы начинаются на MAP.

(MAPCAR fn sp₁ sp₂ ... sp_n)

Возвращает список, состоящий из результатов последовательного применения функционального аргумента (функции n переменных) к соответствующим элементам n списков. Число аргументов-списков должно быть равно числу аргументов функционального аргумента.

Пример 3:

Заменить в списке все числа на пару (<число> *).

```
(mapcar  
  (lambda (x)  
    (cond  
      ((numberp x) (list x '*))  
      (t x)  
    )  
  )  
  '(a 1 2 b c)) → (a (1*) (2*) b c)
```

Пример 4:

Функция **SUM3** вычисляет сумму кубов элементов числового списка.

```
1) (defun sum3(l)
      (apply '+ (mapcar '* l l l)))
    )
```

```
2) (defun sum3(l)
      (eval (cons '+ (mapcar '* l l l))))
    )
```


(**MAPLIST** fn sp₁ sp₂ ... sp_n)

Отображающий функционал **MAPLIST** действует подобно **MARCAR**, но действия осуществляются не над элементами списков, а над последовательными хвостами этих списков, начиная с самих списков.

Пример 5:

(maplist reverse '(a b c d)) → ((d c b a) (c b a) (b a) (a))
(maplist (lambda (l) (apply + l)) '(1 2 3 4 5)) →
→ (15 14 12 9 5)

Объединяющие функционалы MAPCAN и MAPCON

Работа их аналогична соответственно MAPCAR и MAPLIST. Различие заключается в способе построения результирующего списка. Если функционалы MAPCAR и MAPLIST строят новый список из результатов применения функционального аргумента с помощью функции LIST, то функционалы MAPCAN и MAPCON для построения нового списка используют структуроразрушающую псевдофункцию NCONC, которая делает на внешнем уровне то же самое, что и функция APPEND. Функционалы MAPCAN и MAPCON удобно использовать в качестве фильтров для удаления нежелательных элементов из списка.

(MAPCAN fn sp₁ sp₂ ... sp_n)

Пример 6:

Удалить из числового списка все элементы, кроме отрицательных.

```
(mapcan (lambda (x)
  (cond
    ((>= x 0) nil)
    (t (list x))
  )
)
```

'(1 -1 0 2 -3)) → (-1 -3)
(1) (-1) (1) (1) (-3)

(MAPCON fn sp₁ sp₂ ... sp_n)

Пример 7:

Преобразовать одноуровневый список во множество.

```
(mapcon (lambda (l)
  (cond
    ((null l) l)
    ((member (car l) (cdr l)) nil)
    (t (list (car l)))))
  )
)
```

'(1 2 1 3 4 2 5)) → () () (1) (3) (4) (2) (5) →
 (1 3 4 2 5)

