Функциональное и логическое программирование

Лекция 2

1.3.3 Ключевые слова параметров в лямбда-списке

При определении функции можно в лямбда-списке использовать ключевые слова, с помощью которых можно по-разному трактовать аргументы функции при ее вызове.

Ключевое слово начинается символом &, записывается перед параметрами, на которые действует, и его действие распространяется до следующего ключевого слова. Параметры, указанные до первого & обязательны при вызове.

Ключевое слово	Значение ключевого слова
&optional	необязательные параметры

Для необязательных параметров можно указать значение при его отсутствии (по умолчанию nil).

Пример 4:

```
(defun f (x &optional (y 5) z)
   (list x y z)
)
```

Можно обращаться к функции с разным количеством параметров:

```
(f '(a b) '(c) 3) → ((a b) (c) 3)
(f '(a b) '(c)) → ((a b) (c) nil)
(f '(a b)) → ((a b) 5 nil)
```

1.4 Предикаты

Если перед вычислением функции необходимо убедиться, что ее аргументы принадлежат области определения, или возникает задача подсчета элементов списка определенного типа, то используют специальные функции – предикаты.

Предикатом называется функция, которая используется для распознавания или идентификации и возвращает в качестве результата логическое значение – специальные символы t или nil.

Часто имена предикатов заканчиваются на **P** (от слова Predicate).

(**ATOM** s-выражение) – проверяет, является ли аргумент атомом.

```
(atom 'a) → t
(atom nil) → t
(atom '(a b)) → nil
```

(**LISTP** s-выражение) – проверяет, является ли аргумент списком.

```
(listp 'a) → nil
(listp nil) → t
```

(**SYMBOLP** s-выражение) – проверяет, является ли аргумент символом

```
(symbolp 'a) → t
(symbolp 3) → nil
```

(**NUMBERP** s-выражение) – проверяет, является ли аргумент числом.

(numberp a) → { t, если a → числом; nil, иначе

(**NULL** s-выражение) – проверяет, является ли аргумент пустым списком.

(null nil) → t

(null '(a)) → nil

Предикаты для работы с числами:

Проверка на равенство:

(= n_1 … n_m),

где n_i – число или связанная с числом переменная.

(= 3 3 3) → t

Проверка на упорядоченность или попадание в диапазон:

(< n_1 … n_m),

где n_i – число или связанная с числом переменная.

(< 1 x 5) <=> 1 < x < 5

Аналогично определяются предикаты: > ; <= ; >=; /=

Предикат для сравнения s-выражений

(EQUAL s_1 s_2**)** - возвращает значение t, если совпадают внешние структуры s-выражений (аргументов функции).

Пример:

(equal '((a) b c) (cons '(a) '(b c))) → t

1.5 Псевдофункция SETQ

Символы могут обозначать представлять другие объекты.

Связать символ с некоторым значением можно при помощи функции **SETQ**.

(**SETQ** p_1 s_1 … p_n s_n) – возвращает значение последнего аргумента (p_i-символ, s_i-s-выражение).

Это псевдофункция. Побочным эффектом ее работы является связывание символов-аргументов с нечетными номерами со значениями вычисленных s-выражений – четных аргументов.

Все образовавшиеся связи действительны в течение всего сеанса работы с интерпретатором Лиспа.

Пример:

(setq x '(1 2) y 'x z y) → x

x → (1 2) y → x z → x

1.6 Разветвление вычислений

Существует специальная синтаксическая форма – предложение:

(**COND**

(P_1 V_1)

(P_2 V_2)

……

(P_n V_n)

),

где P_i – предикат, V_i – вычислимое выражение.

Вычисление значения **COND**:

Последовательно вычисляются предикаты P_1, P_2, … до тех пор, пока не встретится предикат, возвращающий значение отличное от nil. Пусть это будет предикат P_k. Вычисляется выражение V_k и полученное значение возвращается в качестве значения предложения **COND**. Если все предикаты предложения **COND** возвращают nil, то предложение **COND** возвращает nil.

Рекомендуется в качестве последнего предиката использовать специальный символ t, тогда соответствующее ему выражение будет вычисляться во всех случаях, когда ни одно другое условие не выполняется.

(**COND**

(P_1 V_1)
(P_2 V_2)
……
(t V_n)

)

Допустимо следующие использования:

1. (P_i). Если значение P_i отлично от nil, то **COND** возвращает это значение.
2. (P_i V_{i1}… V_{ik}). Если значение P_i отлично от nil, то **COND** последовательно вычисляет V_{i1}… V_{ik} и возвращает последнее вычисленное значение V_{ik}.

В предикатах можно использовать логические функции: AND, OR, NOT.

В случае истинности предикат **AND** возвращает значение своего последнего аргумента, а предикат **OR** - значение своего первого аргумента, отличного от nil.

1.7 Рекурсия

Функция называется *рекурсивной*, если в определяющем ее выражении содержится хотя бы одно обращение к ней самой (явное или через другие функции).

Работа рекурсивной функции

Когда выполнение функции доходит до рекурсивной ветви, функционирующий вычислительный процесс приостанавливается, а запускается с начала новый такой же процесс, но уже на новом уровне.

Прерванный процесс запоминается, он начнет исполняться лишь при окончании запущенного им нового процесса. В свою очередь, новый процесс так же может приостановиться и т.д. Таким образом, образуется стек прерванных процессов, из которых выполняется лишь последний запущенный процесс.

Функция будет выполнена, когда стек прерванных процессов опустеет.

Ошибки при написании рекурсивных функций:

- ошибочное условие, которое приводит к бесконечной рекурсии;
- неверный порядок условий;
- отсутствие проверки какого-нибудь случая.

Рекурсия хорошо подходит для работы со списками, так как списки могут содержать в качестве элементов подсписки, т.е. иметь рекурсивное строение. Для обработки рекурсивных структур естественно использовать рекурсивные функции.

В Лиспе рекурсия используется также для организации повторяющихся вычислений.

1.7.1 Трассировка функций

Включение трассировки:

(**TRACE** <имя функции>) – возвращает имя трассируемой функции или nil.

Если трассируется несколько функций, то их имена – аргументы **TRACE**.

Если была включена трассировка, то при обращении к функции будут отображаться имена вызываемых функций, их аргументов и возвращаемые значения после вычислений.

Цифрами обозначаются уровни рекурсивных вызовов.

После знака ==> указываются возвращаемые значения соответствующего рекурсивного вызова.

Выключение трассировки:

(UNTRACE)

Если отключается трассировка некоторых функций, то их имена - аргументы **UNTRACE**.

1.7.2 Простая рекурсия

Рекурсия называется *простой*, если вызов функции встречается в некоторой ветви лишь один раз. В процедурном программировании простой рекурсии соответствует обыкновенный цикл.

Виды простой рекурсии:

- рекурсия по значению (рекурсивный вызов определяет результат функции);
- рекурсия по аргументу (результат функции – значение другой функции, аргументом которой является рекурсивный вызов исходной функции).

При написании рекурсивных функций старайтесь условия останова рекурсии ставить в начало, делайте проверку всех возможных случаев. Попытайтесь проговорить алгоритм словами.

Пример 1: Определим функцию **FACT**, вычисляющую факториал.

$0! = 1$

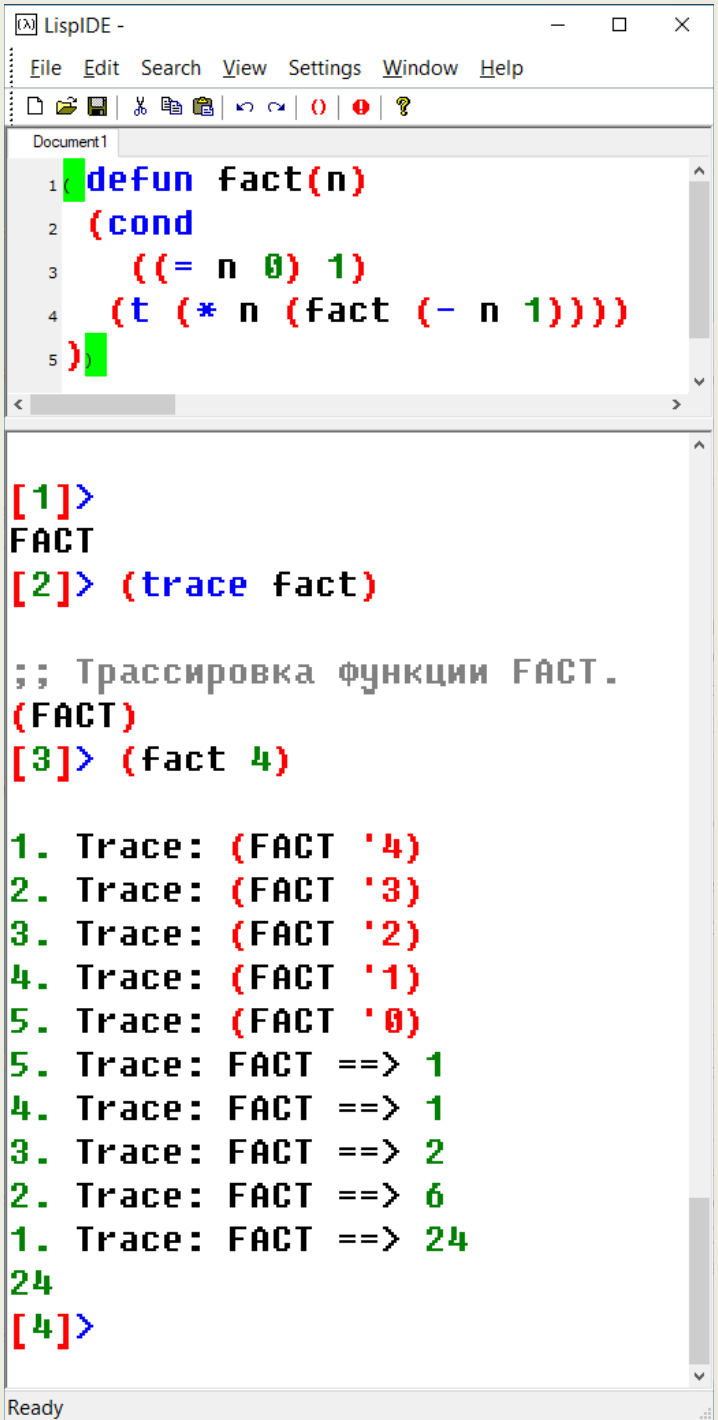
$n! = 1 \cdot 2 \cdot \ldots \cdot (n-1)$

Пусть умеем вычислять $(n-1)!$

Тогда $n! = n \cdot (n-1)!$

$0! = 1$

```
(defun fact (n)
  (cond
    ((= n 0) 1)
    (t (* n (fact (- n 1))))
  )
)
```


LispIDE -
File Edit Search View Settings Window Help
Document1
1 (defun fact(n)
2 (cond
3 ((= n 0) 1)
4 (t (* n (fact (- n 1))))
5)
[1]>
FACT
[2]> (trace fact)
;; Трассировка функции FACT.
(FACT)
[3]> (fact 4)
1. Trace: (FACT '4)
2. Trace: (FACT '3)
3. Trace: (FACT '2)
4. Trace: (FACT '1)
5. Trace: (FACT '0)
5. Trace: FACT ==> 1
4. Trace: FACT ==> 1
3. Trace: FACT ==> 2
2. Trace: FACT ==> 6
1. Trace: FACT ==> 24
24
[4]>
Ready

Пример 2: Определим функцию **COPY**, копирующую список на верхнем уровне (без учета вложенностей).

```
( defun copy (l)
  (cond
    ((null l) l)
    (t (cons (car l)
             (copy (cdr l))
       )
    )
  )
)
```

LispIDE -

File Edit Search View Settings Window Help

Document1

```
(defun copy(L)
 (cond
  ((null L) L)
  (t (cons (car L)(copy (cdr L))))
))
```

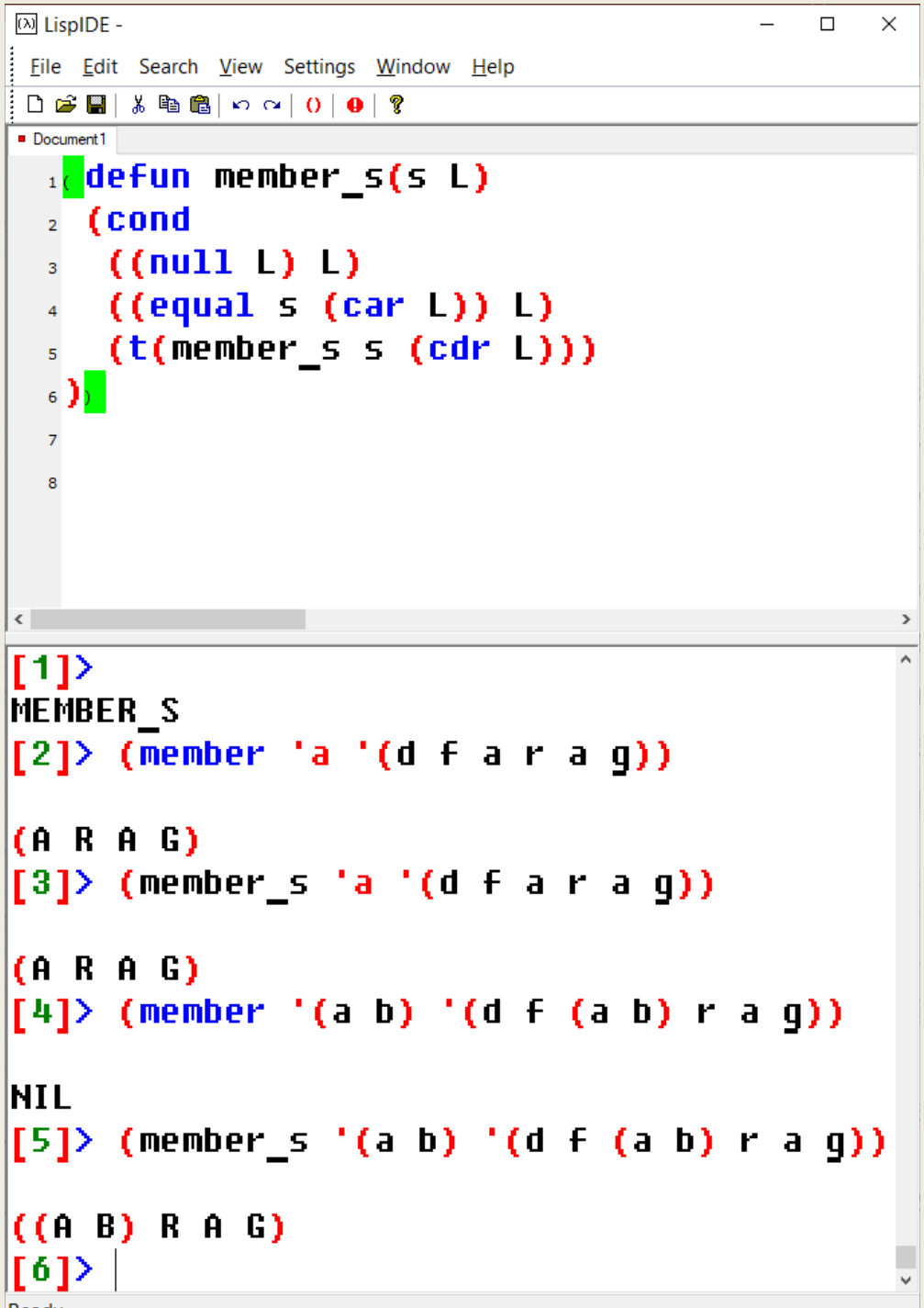
```
[1]>
COPY
[2]> (trace copy)

;; Трассировка функции COPY.
(COPY)
[3]> (copy '(a b c d))

1. Trace: (COPY '(A B C D))
2. Trace: (COPY '(B C D))
3. Trace: (COPY '(C D))
4. Trace: (COPY '(D))
5. Trace: (COPY 'NIL)
5. Trace: COPY ==> NIL
4. Trace: COPY ==> (D)
3. Trace: COPY ==> (C D)
2. Trace: COPY ==> (B C D)
1. Trace: COPY ==> (A B C D)
(A B C D)
[4]>
```

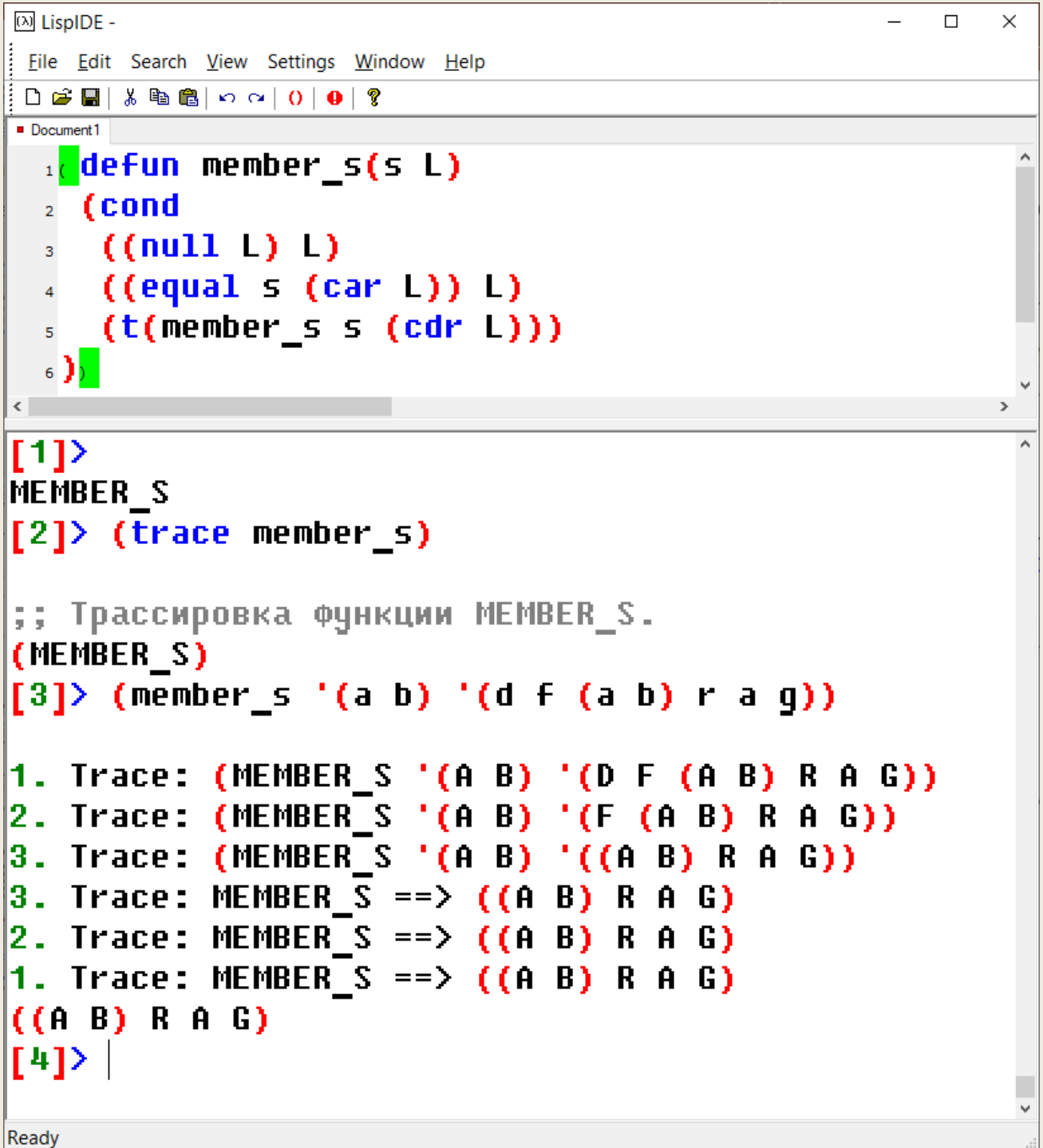
Ready

Пример 3: Определим функцию **MEMBER_S**, проверяющую принадлежность s-выражения списку на верхнем уровне. В случае, если s-выражение принадлежит списку, функция возвращает часть списка, начинающуюся с первого вхождения s-выражения в список.

В Лиспе имеется аналогичная встроенная функция **MEMBER** (но она использует в своем теле функцию **EQ**, поэтому не работает для вложенных списков).

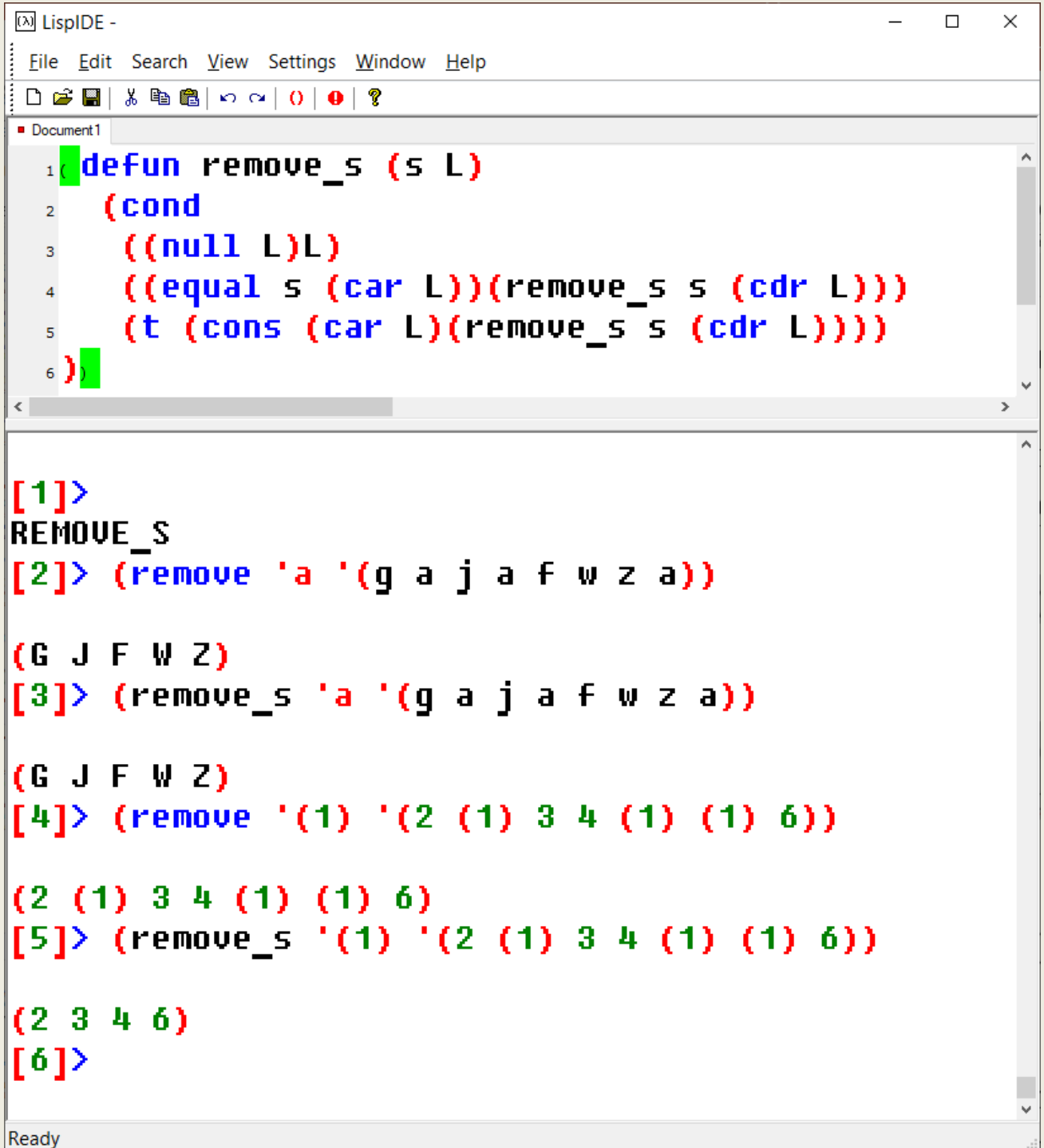
```
(defun member_s (s l)
  (cond
   ((null l) nil)
   ((equal s (car l)) l)
   (t (member_s s (cdr l)))
   )
 )
```


LispIDE -
File Edit Search View Settings Window Help
Document1
1 (defun member_s(s L)
2 (cond
3 ((null L) L)
4 ((equal s (car L)) L)
5 (t(member_s s (cdr L)))
6)
7
8
[1]>
MEMBER_S
[2]> (member 'a '(d f a r a g))
(A R A G)
[3]> (member_s 'a '(d f a r a g))
(A R A G)
[4]> (member '(a b) '(d f (a b) r a g))
NIL
[5]> (member_s '(a b) '(d f (a b) r a g))
((A B) R A G)
[6]>


LispIDE -
File Edit Search View Settings Window Help
Document1
1 (defun member_s(s L)
2 (cond
3 ((null L) L)
4 ((equal s (car L)) L)
5 (t(member_s s (cdr L)))
6))
[1]>
MEMBER_S
[2]> (trace member_s)
;; Трассировка функции MEMBER_S.
(MEMBER_S)
[3]> (member_s '(a b) '(d f (a b) r a g))
1. Trace: (MEMBER_S '(A B) '(D F (A B) R A G))
2. Trace: (MEMBER_S '(A B) '(F (A B) R A G))
3. Trace: (MEMBER_S '(A B) '((A B) R A G))
3. Trace: MEMBER_S ==> ((A B) R A G)
2. Trace: MEMBER_S ==> ((A B) R A G)
1. Trace: MEMBER_S ==> ((A B) R A G)
((A B) R A G)
[4]>
Ready

Пример 4: Определим функцию **REMOVE_S**, удаляющую все вхождения заданного s-выражения в список на верхнем уровне. В Лиспе имеется аналогичная встроенная функция **REMOVE**, но она не работает для вложенных списков.

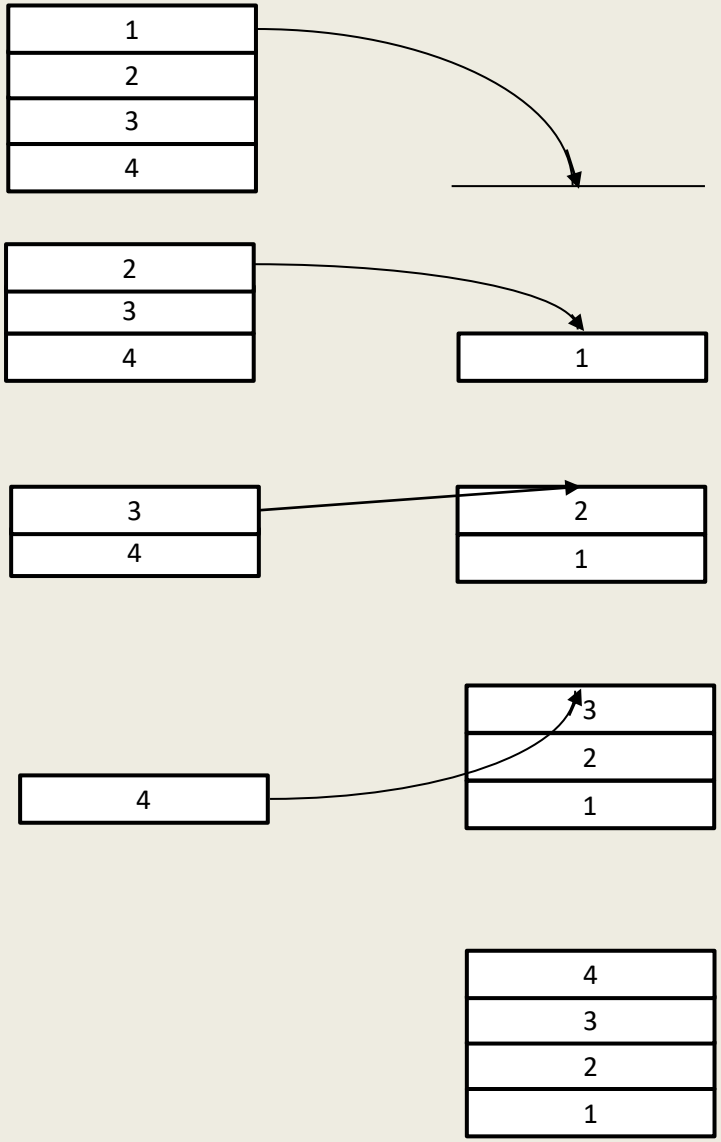
```
(defun remove_s (s l)
  (cond
    ((null l) l)
    ((equal s (car l)) (remove_s s (cdr l)))
    (t (cons (car l) (remove_s s (cdr l))))
  )
)
```


LispIDE -
File Edit Search View Settings Window Help
Document1
1 (defun remove_s (s L)
2 (cond
3 ((null L)L)
4 ((equal s (car L))(remove_s s (cdr L)))
5 (t (cons (car L)(remove_s s (cdr L))))
6)
[1]>
REMOVE_S
[2]> (remove 'a '(g a j a f w z a))
(G J F W Z)
[3]> (remove_s 'a '(g a j a f w z a))
(G J F W Z)
[4]> (remove '(1) '(2 (1) 3 4 (1) (1) 6))
(2 (1) 3 4 (1) (1) 6)
[5]> (remove_s '(1) '(2 (1) 3 4 (1) (1) 6))
(2 3 4 6)
[6]>
Ready

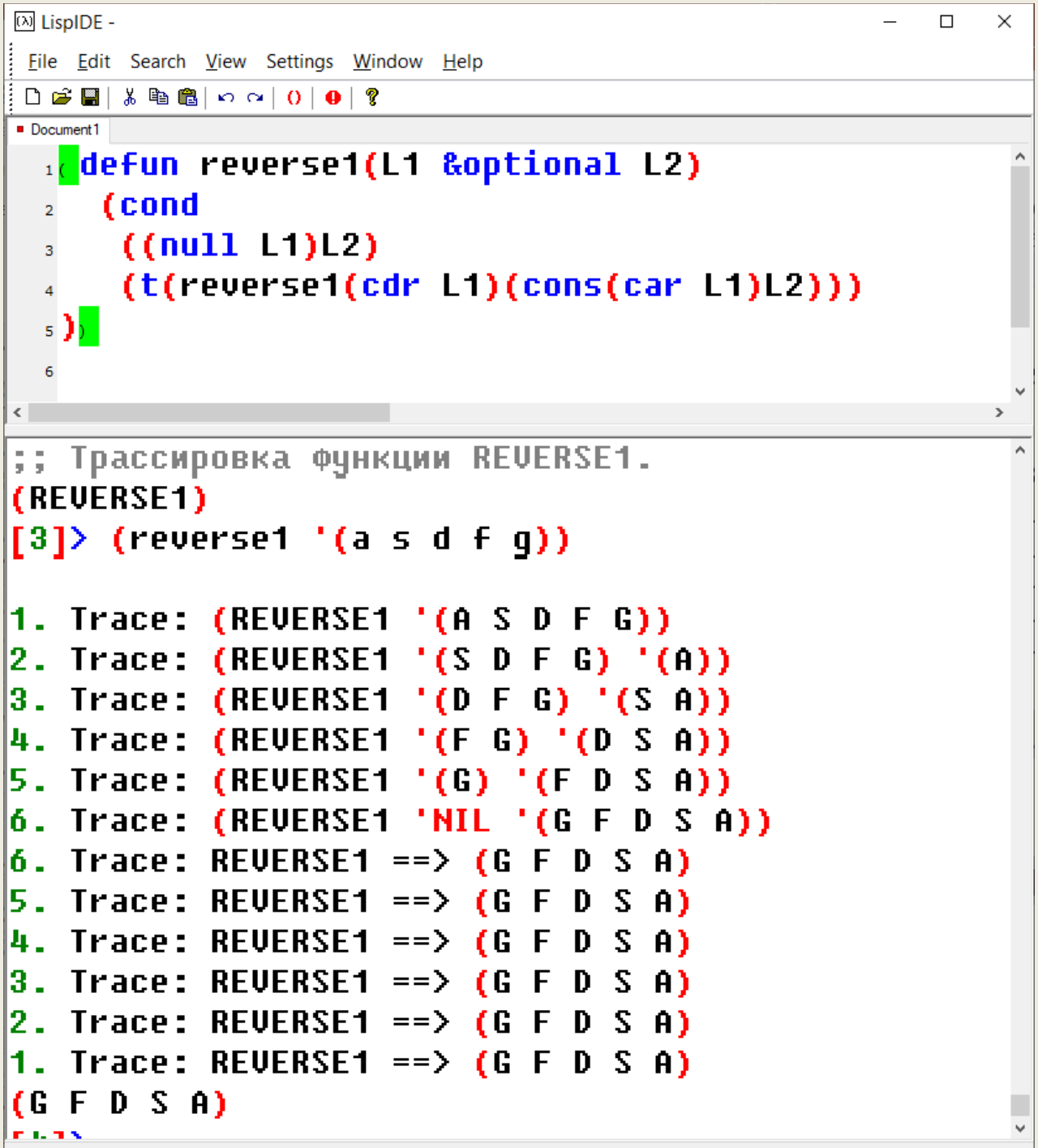
1.7.3 Использование накапливающих параметров

При работе со списками их просматривают слева направо. Но иногда более естественен просмотр справа налево. Например, обращение списка было бы легче осуществить, если бы была возможность просмотра в обратном направлении. Для сохранения промежуточных результатов используют вспомогательные параметры.

Пример 5: Определим **REVERSE1**, обращающую список на верхнем уровне, с дополнительным параметром для накапливания результата обращения списка.

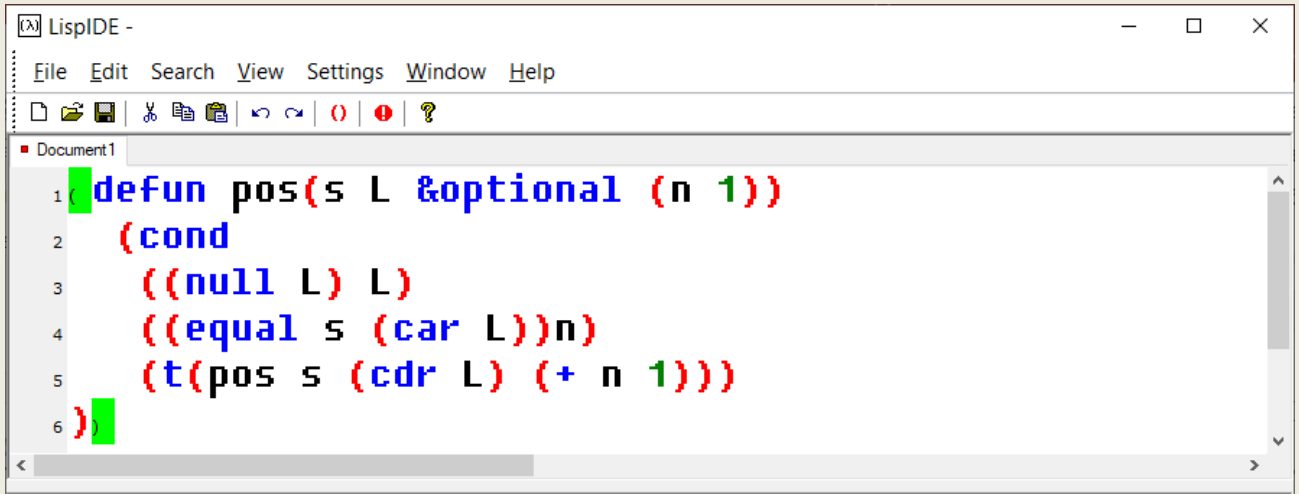


```
(defun reverse1 (l1 &optional l2)
  (cond
    ((null l1) l2)
    (t (reverse1 (cdr l1) (cons (car l1) l2)))
  )
)
```


LispIDE -
File Edit Search View Settings Window Help
Document1
1 (defun reverse1(L1 &optional L2)
2 (cond
3 ((null L1)L2)
4 (t(reverse1(cdr L1)(cons(car L1)L2)))
5)
6
;; Трассировка функции REVERSE1.
(REVERSE1)
[3]> (reverse1 '(a s d f g))
1. Trace: (REVERSE1 '(A S D F G))
2. Trace: (REVERSE1 '(S D F G) '(A))
3. Trace: (REVERSE1 '(D F G) '(S A))
4. Trace: (REVERSE1 '(F G) '(D S A))
5. Trace: (REVERSE1 '(G) '(F D S A))
6. Trace: (REVERSE1 'NIL '(G F D S A))
6. Trace: REVERSE1 ==> (G F D S A)
5. Trace: REVERSE1 ==> (G F D S A)
4. Trace: REVERSE1 ==> (G F D S A)
3. Trace: REVERSE1 ==> (G F D S A)
2. Trace: REVERSE1 ==> (G F D S A)
1. Trace: REVERSE1 ==> (G F D S A)
(G F D S A)

Пример 6: Определим функцию **POS**, определяющую позицию первого вхождения s-выражения в список (на верхнем уровне).

```
(defun pos (s l &optional (n 1))
  (cond
    ((null l) nil)
    ((equal s (car l)) n)
    (t (pos s (cdr l) (+ n 1)))
  )
)
```




```
[3]> (pos 'r '(q w e r t y r a))

1. Trace: (POS 'R '(Q W E R T Y R A))
2. Trace: (POS 'R '(W E R T Y R A) '2)
3. Trace: (POS 'R '(E R T Y R A) '3)
4. Trace: (POS 'R '(R T Y R A) '4)
4. Trace: POS ==> 4
3. Trace: POS ==> 4
2. Trace: POS ==> 4
1. Trace: POS ==> 4
4
[4]> (pos '(q w) '(q w (q w) r (q w) y r a))

1. Trace: (POS '(Q W) '(Q W (Q W) R (Q W) Y R A))
2. Trace: (POS '(Q W) '(W (Q W) R (Q W) Y R A) '2)
3. Trace: (POS '(Q W) '((Q W) R (Q W) Y R A) '3)
3. Trace: POS ==> 3
2. Trace: POS ==> 3
1. Trace: POS ==> 3
3
```

1.7.4 Параллельная рекурсия

Рекурсия называется *параллельной*, если рекурсивный вызов встречается одновременно в нескольких аргументах функции. Такая рекурсия встречается обычно при обработке вложенных списков. В операторном программировании параллельная рекурсия соответствует следующим друг за другом (текстуально) циклам.

Параллельность рекурсии не временная, а текстуальная. При выполнении тела функции в глубину идет сначала левый вызов (рекурсия «в глубину»), а потом правый (рекурсия «в ширину»).

Пример 7: Определим функцию **COPY_ALL**, копирующую список на всех уровнях.

```
(defun copy_all (l)
  (cond
    ((null l) l)
    ((atom l) l)
    (t (cons (copy_all (car l))
             (copy_all (cdr l))))))
```

```
(defun copy_all  (L)
 (cond
  ((null L)  nil )
  ((atom L) L)
  (t  (cons (copy_all (car L))
            (copy_all (cdr L))
      )
  )
 )
)
```

```
[5]> (copy_all '((1)((a)b)))

1. Trace: (COPY_ALL '((1) ((A) B)))
2. Trace: (COPY_ALL '(1))
3. Trace: (COPY_ALL '1)
3. Trace: COPY_ALL ==> 1
3. Trace: (COPY_ALL 'NIL)
3. Trace: COPY_ALL ==> NIL
2. Trace: COPY_ALL ==> (1)
2. Trace: (COPY_ALL '(((A) B)))
3. Trace: (COPY_ALL '((A) B))
4. Trace: (COPY_ALL '(A))
5. Trace: (COPY_ALL 'A)
5. Trace: COPY_ALL ==> A
5. Trace: (COPY_ALL 'NIL)
5. Trace: COPY_ALL ==> NIL
4. Trace: COPY_ALL ==> (A)
4. Trace: (COPY_ALL '(B))
5. Trace: (COPY_ALL 'B)
5. Trace: COPY_ALL ==> B
5. Trace: (COPY_ALL 'NIL)
5. Trace: COPY_ALL ==> NIL
4. Trace: COPY_ALL ==> (B)
3. Trace: COPY_ALL ==> ((A) B)
3. Trace: (COPY_ALL 'NIL)
3. Trace: COPY_ALL ==> NIL
2. Trace: COPY_ALL ==> (((A) B))
1. Trace: COPY_ALL ==> ((1) ((A) B))
```

Ready

Пример 8: Определим функцию **IN_ONE**, преобразующую список в одноуровневый (удаление вложенных скобок).

```
(defun in_one (l)
  (cond
    ((null l) l)
    ((atom l) (list l))
    (t (append (in_one (car l))
               (in_one (cdr l))))
  )
)
```

```
(defun in_one  (L)
 (cond
  ((null L)  nil )
  ((atom L) (list L))
  (t (append (in_one (car L))
             (in_one (cdr L))))
  )
 )
)
```

```
[10]> (in_one '(1 (2 (3))))

1. Trace: (IN_ONE '(1 (2 (3))))
2. Trace: (IN_ONE '1)
2. Trace: IN_ONE ==> (1)
2. Trace: (IN_ONE '((2 (3))))
3. Trace: (IN_ONE '(2 (3)))
4. Trace: (IN_ONE '2)
4. Trace: IN_ONE ==> (2)
4. Trace: (IN_ONE '((3)))
5. Trace: (IN_ONE '(3))
6. Trace: (IN_ONE '3)
6. Trace: IN_ONE ==> (3)
6. Trace: (IN_ONE 'NIL)
6. Trace: IN_ONE ==> NIL
5. Trace: IN_ONE ==> (3)
5. Trace: (IN_ONE 'NIL)
5. Trace: IN_ONE ==> NIL
4. Trace: IN_ONE ==> (3)
3. Trace: IN_ONE ==> (2 3)
3. Trace: (IN_ONE 'NIL)
3. Trace: IN_ONE ==> NIL
2. Trace: IN_ONE ==> (2 3)
1. Trace: IN_ONE ==> (1 2 3)
(1 2 3)
[11]>
Ready
```

Пример 9: Определим функцию **MAX_IN_LIST**, находящую максимальный элемент в числовом списке, содержащем подсписки.

```
(defun max-in-list (l)
  (cond
    ((atom l) l)
    ((null (cdr l)) (car l))
    (t (max (max-in-list (car l))
            (max-in-list (cdr l))))
  )
)
```

```
(defun max_in_list (L)
 (cond
  ((atom L)  L)
  ((null (cdr L)) (max_in_list (car L)))
  (t (max (max_in_list (car L))
          (max_in_list (cdr L)))
  )
 )
)
```

```
[14]> (max_in_list '(((10) 2) 30 ((4))))

1. Trace: (MAX_IN_LIST '(((10) 2) 30 ((4))))
2. Trace: (MAX_IN_LIST '((10) 2))
3. Trace: (MAX_IN_LIST '(10))
4. Trace: (MAX_IN_LIST '10)
4. Trace: MAX_IN_LIST ==> 10
3. Trace: MAX_IN_LIST ==> 10
3. Trace: (MAX_IN_LIST '(2))
4. Trace: (MAX_IN_LIST '2)
4. Trace: MAX_IN_LIST ==> 2
3. Trace: MAX_IN_LIST ==> 2
2. Trace: MAX_IN_LIST ==> 10
2. Trace: (MAX_IN_LIST '(30 ((4))))
3. Trace: (MAX_IN_LIST '30)
3. Trace: MAX_IN_LIST ==> 30
3. Trace: (MAX_IN_LIST '(((4))))
4. Trace: (MAX_IN_LIST '((4)))
5. Trace: (MAX_IN_LIST '(4))
6. Trace: (MAX_IN_LIST '4)
6. Trace: MAX_IN_LIST ==> 4
5. Trace: MAX_IN_LIST ==> 4
4. Trace: MAX_IN_LIST ==> 4
3. Trace: MAX_IN_LIST ==> 4
2. Trace: MAX_IN_LIST ==> 30
1. Trace: MAX_IN_LIST ==> 30
30
[15]>
Ready
```

1.8 Интерпретатор языка Лисп EVAL

Интерпретатор Лиспа называется **EVAL** и его можно так же, как и другие функции вызывать из программы.

«Лишний» вызов интерпретатора может, например, снять эффект блокировки вычисления от функции **QUOTE** или найти значение значения выражения, т.е. осуществить двойное вычисление.

(**EVAL** s-выражение)

Возвращает значение значения аргумента.

Примеры:

1) (setq x '(a b c)) → (a b c) x → (a b c)
   (eval 'x) → (a b c)
   (eval x) → ошибка (нет ф-ции a)

2) (setq a 'b) → b a → b
   (setq b 'c) → c b → c
   (eval a) → c
   (eval 'a) → b

3) (setq x '(1 2 3)) → (1 2 3) x → (1 2 3)
   (eval (cons '+ '(1 2 3))) → 6

Используя **EVAL**, мы можем выполнить «оператор», который создан Лисп-программой и который может меняться в процессе выполнения программы.

Лисп позволяет с помощью одних функций формировать определения других функций, программно анализировать и редактировать эти определения как s-выражения, а затем, используя функцию **EVAL**, исполнять их.