# 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 1) z)
    (list x y z)
)
```

Можно обращаться к функции с разным количеством параметров:

```
(f '(a b) 'c '(d e f)) → ((a b) c (d e f))
(f '(a b) 'c) → ((a b) c nil)
(f '(a b)) → ((a b) 1 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 '(a)) → nil
(atom nil) → t
```

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

```
(listp '(1 2 3)) → t
(listp nil) → t
```

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

```
(symbolp '(a b)) → nil
```

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

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

## Предикаты для работы с числами:

Проверка на равенство:

(= $n_1$ … $n_m$),

где $n_i$ – число или связанная с числом переменная.

Проверка на упорядоченность или попадание в диапазон:

(< $n_1$ … $n_m$),

где $n_i$ – число или связанная с числом переменная.

$1 < x < 5 \quad <=> \quad (< \ 1 \ \ x \ \ 5)$

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 '(1 2) '(1 2)) → t

(eq '(1 2) '(1 2)) → nil

(eql '(1 2) '(1 2)) → nil

} не использовать

# 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 'y y 'z z (+ 1 2)) → 3

x → y → z → 3 – побочный эффект

# 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>)

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.

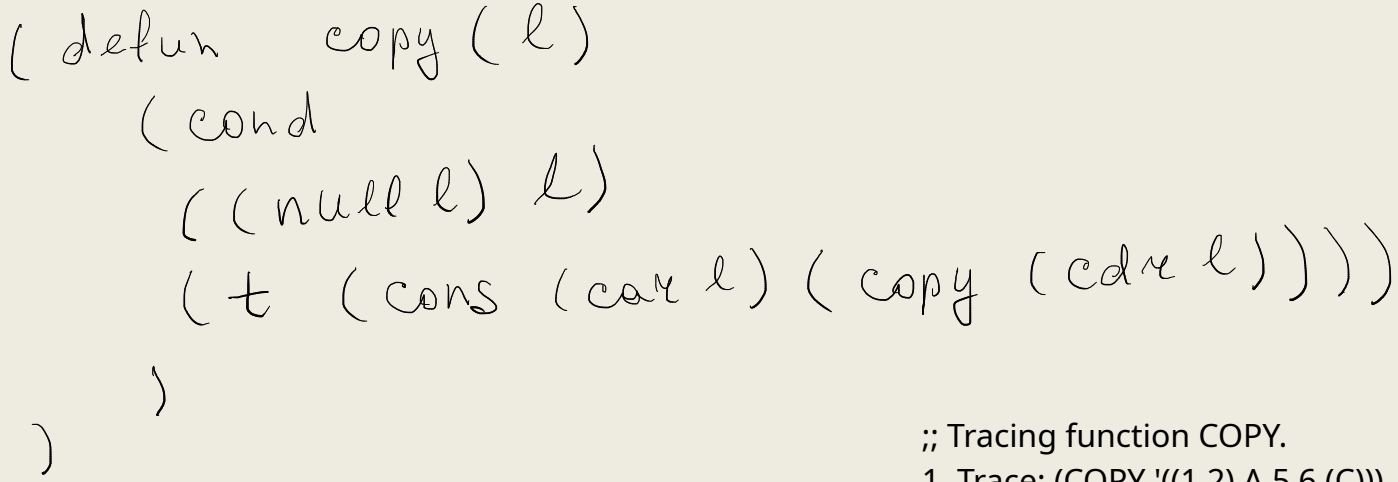
Пример 1: Определим функцию **FACT**, вычисляющую факториал.

$$n! = \begin{cases} 1, & \text{при } n = 0 \\ n * (n-1)! \end{cases}$$

```
(defun fact(n)
  (cond
   ((= n 0) 1)
   (t (* n (fact (- n 1))))
   )
 )
 (trace fact)
(print(fact 5) )
```

```
;; Tracing function FACT.
1. Trace: (FACT '5)
2. Trace: (FACT '4)
3. Trace: (FACT '3)
4. Trace: (FACT '2)
5. Trace: (FACT '1)
6. Trace: (FACT '0)
6. Trace: FACT ==> 1
5. Trace: FACT ==> 1
4. Trace: FACT ==> 2
3. Trace: FACT ==> 6
2. Trace: FACT ==> 24
1. Trace: FACT ==> 120
120
```

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))))
      )
)
 (trace copy)
(print(copy '((1 2) a 5 6 (c))) )
```

```
;; Tracing function COPY.
1. Trace: (COPY '((1 2) A 5 6 (C)))
2. Trace: (COPY '(A 5 6 (C)))
3. Trace: (COPY '(5 6 (C)))
4. Trace: (COPY '(6 (C)))
5. Trace: (COPY '((C)))
6. Trace: (COPY 'NIL)
6. Trace: COPY ==> NIL
5. Trace: COPY ==> ((C))
4. Trace: COPY ==> (6 (C))
3. Trace: COPY ==> (5 6 (C))
2. Trace: COPY ==> (A 5 6 (C))
1. Trace: COPY ==> ((1 2) A 5 6 (C))
((1 2) A 5 6 (C))
```

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) L)
      ((equal s (car L))L)
      (t (member_s s (cdr L)))
      )
)
 ;(trace copy)
(print(member_s '(c) '((1 2) a 5 (c) 6 (c))) )
(print(member '(c) '((1 2) a 5 (c) 6 (c))) )
(print (member_s 'a '(d s a g a h)))
(print (member 'a '(d s a g a h)))
```

```
((C) 6 (C))
NIL
(A G A H)
(A G A H)
```

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.

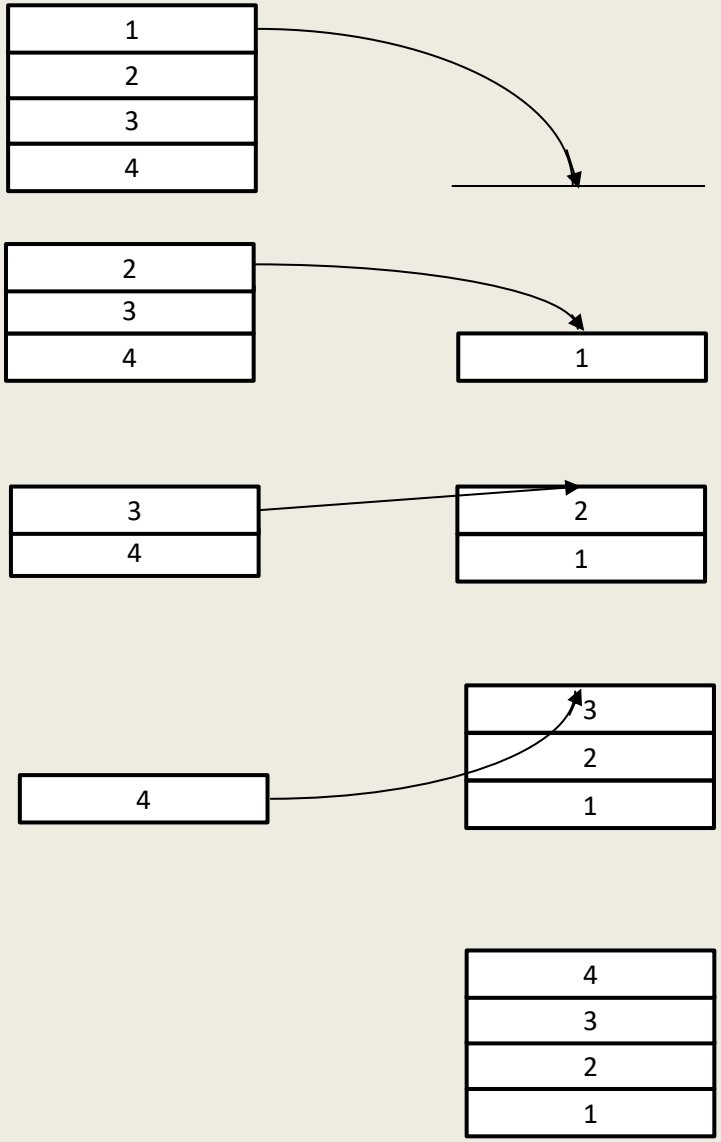
```
(defun remove_S(S L)
   (cond
      ((null L) L)
      ((equal s (car L))(remove_s s (cdr L)))
      (t (cons(car L) (remove_s s (cdr L))))
      )
)

(print(remove_s '(c) '((1 2) a 5 (c) 6 (c))) )
(print(remove '(c) '((1 2) a 5 (c) 6 (c))) )
(print (remove_s 'a '(d s a g a h)))
(print (remove 'a '(d s a g a h)))
```

```
((1 2) A 5 6)
((1 2) A 5 (C) 6 (C))
(D S G H)
(D S G H)
```

## 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)) )
      )
)

(trace reverse1)
(print(reverse1 '(a s d f g)))
```

```
        ;; Tracing function REVERSE1.
   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) L)
      ((equal s (car L)) n)
      (t (pos s (cdr L) (+ n 1)) )
      )
)

(print(pos 'a  '(s h a d f a g)))
```

```
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
      ((atom L)L)
      ((null L) L)
      (t (cons (copy_all (car L))(copy_all (cdr L))) )
      )
)
 (trace copy_all)
(print(copy_all '(s (h (a d)) f )))
```

```
        ;; Tracing function COPY_ALL.
1. Trace: (COPY_ALL '(S (H (A D)) F))
2. Trace: (COPY_ALL 'S)
2. Trace: COPY_ALL ==> S
2. Trace: (COPY_ALL '((H (A D)) F))
3. Trace: (COPY_ALL '(H (A D)))
4. Trace: (COPY_ALL 'H)
4. Trace: COPY_ALL ==> H
4. Trace: (COPY_ALL '((A D)))
5. Trace: (COPY_ALL '(A D))
6. Trace: (COPY_ALL 'A)
6. Trace: COPY_ALL ==> A
6. Trace: (COPY_ALL '(D))
7. Trace: (COPY_ALL 'D)
7. Trace: COPY_ALL ==> D
7. Trace: (COPY_ALL 'NIL)
7. Trace: COPY_ALL ==> NIL
6. Trace: COPY_ALL ==> (D)
5. Trace: COPY_ALL ==> (A D)
5. Trace: (COPY_ALL 'NIL)
5. Trace: COPY_ALL ==> NIL
4. Trace: COPY_ALL ==> ((A D))
3. Trace: COPY_ALL ==> (H (A D))
3. Trace: (COPY_ALL '(F))
4. Trace: (COPY_ALL 'F)
4. Trace: COPY_ALL ==> F
4. Trace: (COPY_ALL 'NIL)
4. Trace: COPY_ALL ==> NIL
3. Trace: COPY_ALL ==> (F)
2. Trace: COPY_ALL ==> ((H (A D)) F)
1. Trace: COPY_ALL ==> (S (H (A D)) F)
(S (H (A D)) F)
```

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))))
   )
)

(print(in_one '(s (h (a d)) f )))
```

(S H A D F)

# 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:

(setq x 'y y 'z z (* 5 6)) → 30

x → y → z → 30

(eval 'x) → y

(eval x) → z

(eval (eval x)) → 30

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.