Лекция №4

Организация вычислений в Лиспе.
Часть 1

Передача параметров и область их действия.

В программировании используются в основном два способа передачи параметров: по значению и по ссылке.

По значению. Изменения значения формального параметра во время вычисления функции никак не отражаются на значении фактического параметра. С помощью таких параметров можно передавать информацию только внутрь процедур, но не обратно из них. В Лиспе используется передача параметров по значению.

По ссылке. Изменения значений формальных параметров видны извне, и можно возвращать данные из процедуры с помощью присваивания значений формальным параметрам. В Лиспе в чистом виде передачи параметров по ссылке не предусмотрено.

Формальные параметры функции называют **статическими переменными** (локальными). Связи статической переменной действительны только в пределах той формы, в которой они определены. После вычисления функции, созданные на это время связи формальных параметров ликвидируются, и происходит возврат к тому состоянию, который был до вызова функции.

(SETQ x 'a) => a

(DEFUN abc (x) (SETQ x ‘b) (PRINT x)) => abc

(abc x) => b => b

x => a – внутри функции значение x изменяется, но первоначальное значение не меняется.

Свободные переменные – это переменные, которые используются в функции, но не входят в число ее формальных параметров. Изменения свободных переменных остаются в силе после окончания выполнения функции.

(SETQ x 'r) => r

x => r

(DEFUN abc1 () (SETQ x ‘b)) => abc1

(abc1) => b

x => b – при вызове функции авс1 значение переменной x изменилось

Для задания **динамических (глобальных) переменных** используется директива

(DEFVAR переменная &OPTIONAL начальное значение)

Значение глобальной переменной определяется динамически во время вычисления, а не в зависимости от места ее определения. Значение свободной переменной, являющейся формальным параметром внешнего вызова, вычисляется следующим образом:

- использование статической переменной в качестве формального параметра во внешней форме не влияет на значение свободной переменной. Значение свободной переменной определяется в соответствии с глобальным значением, присвоенным на самом внешнем уровне с помощью SETQ;
- для динамической переменной связь, установленная в более внешнем вызове, остается в силе для всех вложенных контекстов, возникающих во время вычисления, если переменная снова не связывается.

(SETQ x 100) => 100 - глобальное значение x

(DEFUN f1 (x) (f2 2)) => f1 - статическая переменная x

(DEFUN f2 (y) (LIST x y)) => f2 - свободная переменная x, ее значением является глобальное значение 100.

(f1 1) => (100 2)

(DEFVAR x 100) => x - начиная с этого момента x определяется динамически по последней связи.

(f1 1) => (1 2)

Программа, написанная на языке Лисп, состоит из форм и функций.

Под **формой** понимается такое символьное выражение, значение которого может быть найдено интерпретатором.

К формам относят:

1. самоопределенные формы. Это объекты, представляющие лишь сами себя: числа, константы T и NIL, а также знаки, строки, битовые векторы, ключи, начинающиеся с двоеточия и определяемые через ключевое слово &KEY в лямбда-списке;
2. символы, использующиеся в качестве переменных;
3. формы в виде списочной структуры, которыми являются:

вызовы функций и лямбда-вызовы;

специальные формы: SETQ, QUOTE и другие для управления вычислениями и контекстом,

макровызовы.

У каждой формы свой синтаксис и семантика, основанные на едином способе записи и интерпретации.

Управляющие структуры в языке Лисп выглядят внешне как вызовы функций:

- они записываются в виде скобочного выражения, первый элемент которого действует как имя управляющей структуры, а остальные элементы как аргументы;
- результатом вычисления, как и у функции, является значение, т.е. управляющие структуры представляют собой формы.

Однако управляющие структуры не являются вызовами функций, разные предложения используют аргументы по-разному.

Синтаксические формы по их использованию можно разделить на следующие группы:

- Работа с контекстом:
- QUOTE или блокировка вычисления;
- Вызов функции и лямбда –вызов;
- Предложения LET и LET*.

- Последовательное исполнение:
- PROG1, PROG2 и PROGN.

- Разветвление вычислений:
- условные предложения: COND, IF, WHEN, UNLESS;
- выбирающее предложение CASE.

- Итерации:
- циклические предложения DO, DO*, LOOP, DOTIMES, DOUNTIL.

- Передача управления:
- предложения PROG, GO, RETURN.

- Динамическое управление вычислением:
- THROW, CATCH, BLOCK.

Последовательные вычисления

Предложение LET создает локальную связь внутри формы:
(LET ((m1 знач1) (m2 знач2)...) форма1 форма2 ...).
Предложение LET вычисляется по следующему алгоритму:

- статические переменные m1, m2, ... связываются (одновременно) с соответствующими значениями знач1, знач2, ... ;
- слева направо вычисляются значения формы1, формы2, ... ;
- значение последней формы возвращается в качестве значения всей формы LET:
- после вычисления связи статических переменных ликвидируются и любые их изменения извне не будут видны.

Предложения LET можно делать вложенными одно в другое:
(LET ((x ‘a) (y ‘b)) (LET ((z ‘c)) (LIST x y z))) => (a b c)
(LET ((x (LET ((z ‘a)) z)) (y ‘b)) (LIST x y)) => (a b)
(LET ((x 1) (y (+ x 1))) (LIST x y)) => ERROR

форма LET* вычисляет значения переменных последовательно:
(LET* ((m1 знач1) (m2 знач2)...) форма1 форма2 ...).
(LET* ((x 1) (y (+ x 1))) (LIST x y)) => (1 2)
Продемонстрируем, что связи переменных формы LET восстанавливаются после выхода из формы:
(SETQ x 2) => 2
(LET ((x 0)) (SETQ x 1)) => 1
x => 2

Последовательные вычисления

Предложения **PROG1** и **PROGN** позволяют работать с несколькими вычисляемыми формами:
(PROG1 *форма1 ... формаN*)
(PROGN *форма1 ... формаN*).

Эти специальные формы последовательно вычисляют свои аргументы и в качестве значения возвращают значение первого (PROG1) или последнего (PROGN) аргумента.
Это формы для вычислений, поэтому связи переменных, возникшие при выполнении форм, сохраняются после выхода из PROG:
(PROG1 (SETQ x 1) (SETQ y 5)) => 1
(PROGN (SETQ j 8) (SETQ z (+ x j))) => 9

Условные выражения

- Функция **cond** (обращение к функции cond наз. условным выражением).
- Общий вид обращения к функции cond:
  - (**cond** (***p1 s1***) (***p2 s2***) …(***pn sn***)),
  - где ***p1, …, pn*** – логические выражения,
  - ***s1, …, sn*** -- произвольные выражения.
- Функция **cond** имеет произвольное число аргументов. Каждый из Аргументов – пара, т.е. список из двух элементов. Первый элемент пары – условие. Второй элемент – выражение.
- Функция **cond** принимает одно из значений ***s1, … , sn***.
- Выбор происходит следующим образом: просмотр выполняется слева направо до тех пор, пока не встретится аргумент (***pi, si***) со значением ***pi***, отличным от **nil**. Тогда **cond** возвращает значение ***si***.
- Если все значения ***pi*** равны **nil**, то **cond** **nil** (в некоторых реализациях может быть ошибка).
- Последним выражением ***pn*** часто ставят константу **t**. Это условие всегда удовлетворено (аналог default case).

Пример условного выражения

Рассмотрим условное выражение на алголоподобном языке:

```
y :=  if x>50
          then 10
      else if x > 20
          then 30
      else if x > 10
          then 20
       else 0;
```

Его аналогом является условное выражение на Лиспе:

```
(cond    ((> x 50) 10)
         ((> x 20) 30)
         ((> x 10) 20)
         (t 0)
)
```

Пример 2: Абсолютное значение

```
(defun abs(x)
          (cond ((> 0 x) (- 0 x))
                ( t x))
)
```

Значение функции COND определяется следующим образом:

1. Вычисляются последовательно слева направо значения выражений pi до тех пор, пока не встретится выражение, значение которого отлично от NIL, что интерпретируется как ИСТИНА.
2. Вычисляется результирующее выражение, соответствующее этому предикату, и полученное значение возвращается в качестве значения функции COND.
3. Если истинного предиката нет, то значением COND будет NIL.

В условном выражении может отсутствовать результирующее выражение pi или на его месте часто может быть последовательность выражений:
(COND (p1 e11)...(pi)...(pk ek2...ekn)...).

Если условию не ставится в соответствие результирующее выражение, то в качестве результата предложения COND при истинности предиката выдается само значение предиката. Если же условию соответствует несколько выражений, то при его истинности выражения вычисляются последовательно слева направо и результатом функции COND будет значение последнего выражения последовательности.

Пример условного выражения

Определим логические действия логики высказываний **И** и **ИЛИ**:

```
(DEFUN i (x y)
        (COND (x y)
              (T NIL)))  => I

(i  T  NIL)  =>  NIL


(DEFUN ili (x y)
          (COND (x T)
                (T y)))  =>  ili

(ili  T  NIL)  =>  T
(ili NIL NIL)  =>  NIL
```

Условные выражения

(IF условие то-форма иначе-форма).
Примеры:
(PROGN (SETQ X 1)(IF (> X 0) 'X>0 'X<0)) => X>0
(PROGN (SETQ X -1)(IF (> X 0) 'X>0 'X<0)) => X<0
(IF (> X 0) (SETQ Y (+ Y X)) (SETQ Y (- Y X))) - к значению у прибавляется абсолютное значение x

Вычисление формы, если условие выполнено:
(WHEN условие форма1 форма2 ...).
(PROGN (SETQ x '(c d)) (WHEN (NOT (ATOM x)) (CONS x x))) =>
((C D) C D)

Вычисление формы, если условие не выполнено:
(UNLESS условие форма1 форма2 ...).
(PROGN (SETQ x (READ)) (UNLESS (< x 0) (PRINC “OK”)))
7 - ввод
OK – значение формы
"ok" – вывод

Условные выражения

Выбирающее предложение CASE:

(CASE ключ
(список-ключей1 m11 m12 ...)
(список-ключей2 m21 m22 ...)
....)

Сначала вычисляется значение ключевой формы - ключ. Затем его сравнивают с элементами списка-ключей. Когда в списке найдено значение ключевой формы, начинают вычисляться соответствующие формы mi1, mi2, ... Значение последней формы возвращается в качестве значения всего предложения CASE.

(SETQ ключ 3) => 3
(CASE ключ
(1 ‘ONE)
(2 ‘(ONE + ONE) ‘TWO)
(3 ‘(TWO + ONE) ‘THREE) => THREE
(SETQ k 1) => 1
(CASE x (2 3 4 (SETQ y 2) (SETQ z 3)) (1 2 3 (SETQ y 3) (SETQ z 4))) => 4
(CASE x (k (SETQ y 3)) (T NIL)) => NIL
(SETQ x 'k) => к
(CASE x (k (SETQ y 3)) (T NIL)) => 3

Простые циклы

- Для организации циклических действий используется функция **do** след. формата:

```
(do
    ((var_1 value_1) (var_2 value_2) … (var_n value_n))
    (condition   form_yes_1 form_yes_2 … form_yes_m)
    form_no_1 form_no_2 … form_yes_k
)
```

- Предложение do работает следующим образом: первоначально переменным var_1, var_2, …, var_n присваиваются значения value_1, value_2, …, value_n (параллельно, как в предложении let). Затем проверяется условие выхода из цикла condition.
- Если условие выполняется, последовательно вычисляются формы form_yes_1, form_yes_2, …, form_yes_m, и значение последней вычисленной формы form_yes_m возвращается в качестве значения всего предложения do. Если же условие condition не выполняется, последовательно вычисляются формы form_no_1, form_no_2, …, form_yes_k, и вновь выполняется переход в проверке условия выхода из цикла condition.

- Форма **DO*** последовательно вычисляет свои переменные.

Пример цикла

для возведения x в степень n с помощью умножения определена функция power с двумя аргументами x и n: x – основание степени, n – показатель степени.

```
> (defun power (x n)
    (do
      ((result 1))       ;присваивание начального значения переменной result
      ((= n 0) result) ;условие выхода их цикла
      (setq result (* result x)) (setq n (- n 1)) ;повторяющиеся действия
    )
)
POWER

> (power 2 3)
8
```

(DOTIMES (var count-form result form) форма1 форма2..)

DOTIMES вычисляет COUNT-FORM, значение должно быть целым. Если count-form равно 0 или отрицательно цикл не выполняется. Цикл выполняется от 0 до (count-form-1). Переменная цикла var связывается со значением count-form. После выполнения цикла связь var восстанавливается. Если нет результирующей формы выводится NILL.

Пример.

```
CL-USER 1 > (DOTIMES (x (+ 1 2) 3 ) (PRINT x))

0
1
2
3

CL-USER 2 > (DOTIMES (x (+ 1 2)  ) (PRINT x))

0
1
2
NIL
```

(DOLIST(var list-форма [result-форма]) форма1 форма2…)

Форма DOLIST используется, если цикл надо выполнить с каждым элементом списка:

Пример.

```
>(DOLIST (x '((+ 1 2) (- 1 2) (* 1 2) (/ 1 2))) (PRINT (eval x)))

3
-1
2
1/2
NIL
```

(LOOP m1 m2 … mn)

Бесконечный цикл:
Циклически вычисляются формы m1, …, mn.
Выход из цикла с помощью оператора RETURN.

```
(LOOP (SETQ x (READ)) (COND ((EQUAL x 'end) (RETURN  'end)) (T NIL)) (PRIN1 x))
a A b B end
END
```

LOOP предоставляет язык специального назначения для написания конструкций итерирования.

Можно делать в **LOOP** следующее:

* Итерировать переменную численно или по различным структурам данных.
* Собирать, подсчитывать, суммировать, искать максимальное и минимальное значения по данным, просматриваемым во время цикла.
* Выполнять произвольные выражения Lisp.
* Решать, когда остановить цикл.

•Осуществлять определенные действия при заданных условиях.

•**LOOP** предоставляет синтаксис для следующего:

* Создание локальных переменных для использования внутри цикла.
* Задание произвольных выражений Lisp для выполнения перед и после цикла.

Подсчитывающие циклы

Управление итерированием начинаются с ключевого слова **loop for** или его синонима **as**, за которыми следует имя переменной. Подвыражения (subclauses) предложений **for** могут итерировать по следующему:

* Численные интервалы, вверх или вниз.
* Отдельные элементы списка.
* cons-ячейки, составляющие список.
* Элементы вектора, включая подтипы, такие как строки и битовые векторы.
* Пары хэш-таблицы.
•Результаты повторных вычислений заданной формы

Для указания правил итерирования используются обороты *откуда* (**from**, **downfrom** или **upfrom**), оборот *докуда* (**to**, **upto**, **below**, **downto** или **above**) и оборот *по сколько* .
С upto и downto цикл завершится (без выполнения тела) когда переменная перейдет точку останова; с **below** и **above** он завершится на итерацию ранее.

Оборот *по сколько* состоит из предлога **by** и формы, которая должна вычисляться в положительное число. Переменная будет изменяться на эту величину на каждой итерации, или на единицу, если оборот опущен.

Примеры

```
> (loop for i upto 10 collect i)
(0 1 2 3 4 5 6 7 8 9 10)

> (loop for i from 0 downto -10 collect i)
(0 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10)

 > (loop for i downfrom 20 to 10 collect i)
(20 19 18 17 16 15 14 13 12 11 10)
```

Организация циклов по спискам (коллекциям)

Предложения **for** для итерирования по спискам поддерживают только два предложных оборота: **in** и **on**: for var in list-form

```
 > (loop for i in ‘(10 20 30 40) collect i)  ; поэлементно
(10 20 30 40)

> (loop for i in (list 10 20 30 40) by #'cddr collect i)  ; поэлементно
(10 30)

> (loop for x on (list 10 20 30) collect x)   ; на «хвостах»
((10 20 30) (20 30) (30))


>(loop for (a b) in '((1 2) (3 4) (5 6))
     do (format t "a: ~a; b: ~a~%" a b))
a: 1; b: 2
a: 3; b: 4
a: 5; b: 6
NIL
```

Equals-Then
итерирование

(loop for var = initial-value-form [then step-form] ...)

```
>(loop repeat 5
  for x = 0 then y
  for y = 1 then (+ x y)
  collect y)
 (1 2 4 8 16)
```

Каждое предложение **for** вычисляется отдельно в порядке своего появления. Поэтому в предыдущем цикле на второй итерации x устанавливается в значение y до того, как y изменится (другими словами, в 1). Но y затем устанавливает в значение суммы своего старого значения (все еще 1) и нового значения x.

```
>(loop repeat 5
     for y = 1 then (+ x y)
     for x = 0 then y
     collect y)
(1 1 2 4 8)
```

Локальные переменные в LOOP

with var [= value-form] (помимо переменных цикла в **for)**

Имя ***var*** станет именем локальной переменной, которая перестанет существовать после завершения цикла. Если предложение **with** содержит часть **= value-form**, то перед первой итерацией цикла переменная будет проинициализирована значением ***value-form*.**
Взаимно независимые переменные могут быть объявлены в одном предложении **with** с использованием **and** между такими декларациями

Накопление значения

Каждое предложение накопления начинается с глагола и следует следующему образцу:
verb form [into var]
Каждый раз, при прохождении цикла, предложение накопления вычисляет form и сохраняет значение способом, определяемым глаголом verb. С подпредложением into значение сохраняется в переменную под именем var. Переменная является локальной в цикле, как если бы она была объявлена в предложении with. Без подпредложения into предложение накопления накапливает значения в переменную по умолчанию для всего выражения цикла.

Возможными глаголами являются **collect**, **append**, **nconc**, **count**, **sum**, **maximize** и **minimize**.

Примеры

```
>(loop for i upto 10 sum i)
55
```

```
>(loop for i upto 10 maximize i)
10
```

```
>(loop for i upto 10 count i)
11
```

```
>(loop for i in '((1) (2) (3))  append  i)
(1 2 3)
```

Определим переменную содержащую список случайных чисел

```
>(defparameter  r   (loop  repeat 10 collect (random 10000)))
R
>r
(1622  9799  7925  4608  3427  2344  2743  5998  9775  3524)
```

Следующий цикл вернет список, содержащий различную сводную информацию о числах из r:

```
>(loop for i in r
counting (evenp i) into evens
counting (oddp i) into odds
 summing i into total
maximizing i into max
 minimizing i into min
finally (return (list min  max total evens odds)))
 (1622  9799  51765  5  5)
```

Безусловное выполнение

При выполнении произвольного кода внутри тела цикла используется предложение **do**. Предложение **do** состоит из слова do (или **doing**), за которым следует одна или более форм Lisp, которые вычисляются при вычислении предложения do. Предложение do заканчивается закрывающей скобкой цикла loop или следующим ключевым словом loop. Предложения **do** и **return** вместе называются предложениями безусловного выполнения.

```
>(loop for i from 1 to 10 do (print i))
```

Условное выполнение

IF и **WHEN, unless**

```
>(loop for i from 1 to 10 do (when (evenp i) (print i)))
2
4
6
8
10
NIL
>(loop for i from 1 to 10 when (evenp i) sum i)
30
```

Начало и окончание цикла LOOP

Слова **initially** или **finally** эти предложения включают все формы Lisp до начала следующего предложения цикла либо до его конца. Все формы **initially** комбинируются в единую вводную часть (prologue), которая запускается однократно непосредственно после инициализации всех локальных переменных цикла и перед его телом. Формы **finally** схожим образом комбинируются в заключительную часть (epilogue) и выполняются после последней итерации цикла. И вводная, и заключительная части могут ссылаться на локальные переменные цикла. Предложения **always**, **never** и **thereis** останавливают цикл жестко: они приводят к немедленному возврату из цикла, пропуская не только все последующие предложения **loop**, но и заключительную часть.

```
>(loop for i from 1 to 10 initially (print 'begin) do (when (evenp i) (print i)) finally (print 'end))
BEGIN
2
4
6
8
10
END
NIL
```

Общие правила использования LOOP

Можно комбинировать все выше обсужденные предложения, следуя следующим правилам:

* Предложение **named**, если указывается, должно быть первым предложением.
* После предложения named идут все остальные предложения initially, with, for и repeat.
* Затем идут предложения тела: условного и безусловного выполнения, накопления, критериев завершения.
- Завершается цикл предложениями finally.

Макрос LOOP раскрывается в код, который осуществляет следующие действия:

* Инициализирует все локальные переменные цикла, которые объявлены в предложениях with или for, а также неявно созданные предложениями накопления. Начальные значения форм вычисляются в порядке появления соответствующих предложений в цикле.
* Выполняет формы, предоставляемые предложениями initially (вводная часть), в порядке их появления в цикле.
* Итерирует, выполняя тело цикла.
* Выполняет формы, предоставляемые предложениями finally (заключительная часть), в порядке их появления в цикле.