Лекция №5

Организация вычислений в Лиспе.
Часть 2
Рекурсия. Функционалы.

Передача управления

(PROG (m1 m2 ... mn) или PROG (m1 значение ... mn значение)
форма_1 или метка_1
...
форма_n или метка_n)

Это предложение называют PROG-механизмом с метками перехода

Переменные mi вычисляются одновременно - это локальные статические переменные формы, которые можно использовать для хранения промежуточных результатов. Перед вычислениями им присваивается значение NIL. Если переменных нет, то на месте списка переменных нужно ставить NIL.

PROG* - также как и PROG, но переменные mi вычисляются последовательно.

Метка перехода (метка_i) является символом или целым числом. На такую метку можно передать управление оператором GO:

(GO метка) ;; GO не вычисляет значение своего «аргумента».

Кроме этого, в PROG-механизм входит оператор окончания вычисления и возврата значения:

(RETURN результат)

Операторы предложения PROG вычисляются слева направо (сверху вниз), пропуская метки перехода:

- оператор RETURN прекращает выполнение предложения PROG;
- в качестве значения всего предложения возвращается значение аргумента оператора RETURN;

- если во время вычисления оператор RETURN не встретился, то значением PROG после вычисления его последнего оператора станет NIL;
- после вычисления предложения значения статических переменных возвращаются к своим исходным значениям.

Примеры.

```
(SETQ a 1)
=>  1
(PROG ((a 2) (b a)) (RETURN (if (= a b) '= '/=)))
=>  /=

(PROG* ((a 2) (b a)) (RETURN (if (= a b) '= '/=)))
=>  =
(PROG () 'no-return-value)
=>  NIL
```

Вычисление степени числа. Функция спрашивает число, степень и выводит результат:

```
> (DEFUN stepen ()
(PROG (x n r)
   (print  “основание: ”)   (setq x (read))
   (print  “показатель: ”)   (setq n (read))
   (setq r x )    (if (= n 0) (return 1))
   metka  (if (= n 1)(return r))  (setq r (* r x)) (setq n (- n 1))  (go metka) ) )
STEPEN

CL-USER 13 : 2 > (stepen)

"основание: " 3

"показатель: " 5
243
```

Динамическое управление из другого контекста

До сих пор рассматривались структуры, вычисление которых проводится в одном статическом контексте. В некоторых случаях возникает необходимость прекратить вычисление функции динамически из другого контекста и выдать результат в более раннее состояние, не осуществляя нормальную последовательность возвратов из всех вложенных вызовов. Такая необходимость может возникнуть, например, при обнаружении ошибки. Динамическое прерывание вычисления можно запрограммировать с помощью форм CATCH (поймать) и THROW (бросить):

(CATCH метка форма1 форма2 …) (THROW метка значение)

Выполняется следующая последовательность операций:

- вычисляется метка;
- вычисляются формыі слева направо;
- если не встретилась форма THROW, значение последней выполненной формы возвращается в качестве значения формы CATCH;
- если встретилось форма THROW и ее аргумент «метка» равен метке в форме CATCH, то управление передается в CATCH и значением формы становится значение аргумента THROW.

Пример:

(CATCH 'dummy-tag 1 2 (THROW 'dummy-tag 3) 4) => 3
(CATCH 'dummy-tag 1 2 3 4) => 4
(CATCH (SETQ x 5) 1 2 3 (THROW (+ 1 4) 8) 9 0) = > 8 - метки вычисляются в отличие от формы GO.

(SETQ x (CATCH (SETQ x 5) 1 2 3 (THROW (+ 1 4) 8) 9 0)) => 8
x =>8

Управление передается внутри формы, а не завершается выходом из нее, как при использовании RETURN.

Форма **BLOCK** устанавливает блок с именем name, затем последовательно выполняет формы подобно PROGN:

(BLOCK name form1 ….formn)
(RETURN-FROM name форма-значение)

Если не встретилось RETUTN-FROM, возвращается значение последней формы. Если RETURN-FROM не содержит формы значения, возвращается NILL, в противном случае – значение формы RETURN-FROM.

Формы BLOCK и RETURN-FROM работают совместно, обеспечивая выход на более высокий уровень. В любой точке любой из форм, входящих в блок, может быть использована функция RETURN-FROM, чтобы передать управление из BLOCK -формы и значение формы.

Исключение составляет случай, если определен внешний BLOCK с тем же именем, что и внутренний. В этом случае управление передается внутреннему блоку.

Примеры.

- Блок без передачи управления
  (BLOCK stop (+ 1 2) (+ 3 4)) => 7

- Блок с передачей управления
  (SETQ x 1)
  (BLOCK stop (SETQ x 2) (RETURN-FROM stop) (SETQ x 3)) => NILL - без передачи результата
  x => 2

  (BLOCK stop (RETURN-FROM stop (+ 1 2)) (+ 3 4)) => 3 - с передачей результата

- Выход на верхний уровень
  (BLOCK outer (BLOCK inner (RETURN-FROM outer 1)) 2) => 1

- Выход из внутреннего блока и нормальное завершение внешнего блока

(block twin (block twin (return-from twin 1)) 2) => 2

Рекурсия

Функция является рекурсивной, если в ее определении содержится вызов этой же функции. Рекурсия является простой, если вызов функции встречается в некоторой ветви лишь один раз. Простой рекурсии в процедурном программировании соответствует обыкновенный цикл.

Например, задача нахождения значения факториала n! сводится к нахождению значения факториала (n-1)! и умножения найденного значения на n.

Пример: нахождение значения факториала n!.

```
> (defun factorial (n)
    (cond
      ((= n 0) 1)                ;факториал 0! равен 1
      (t (* (factorial (- n 1)) n)) ;факториал n! равен (n-1)!*n
    )
)
FACTORIAL
```

Отладка / трассировка

Для отладки программы можно использовать возможности трассировки. Трассировка позволяет проследить процесс нахождения решения.

Для того чтобы включить трассировку можно воспользоваться функцией **trace**. После ввода этой директивы интерпретатор будет распечатывать имя функции и значения аргументов каждого вызова трассируемой функции и полученный результат после окончания вычисления каждого вызова.

Например, вызовем трассировку определенной выше функции:

```
> (trace factorial)
> (factorial 3)
Entering: FACTORIAL, Argument list: (3)
    Entering: FACTORIAL, Argument list: (2)
      Entering: FACTORIAL, Argument list: (1)
            Entering: FACTORIAL, Argument list: (0)
            Exiting: FACTORIAL, Value: 1
      Exiting: FACTORIAL, Value: 1
    Exiting: FACTORIAL, Value: 2
Exiting: FACTORIAL, Value: 6
6
```

Для отключения трассировки можно воспользоваться функцией **untrace**:

```
> (untrace factorial)
NIL
```

Виды рекурсии

Можно говорить о двух видах рекурсии: рекурсии по значению и рекурсии по аргументу. Рекурсия по значению определяется в случае, если рекурсивный вызов является выражением, определяющим результат функции. Рекурсия по аргументу существует в функции, возвращаемое значение которой формирует некоторая нерекурсивная функция, в качестве аргумента которой используется рекурсивный вызов.

Приведенный выше пример рекурсивной функции вычисления факториала является примером рекурсии по аргументу, так как возвращаемый результат формирует функция умножения, в качестве аргумента которой используется рекурсивный вызов.

… (t (* (factorial (- n 1)) n)) …

Примером рекурсии по значению м.б. определение принадлежности элемента списку:

```
> (defun member (el list)
    (cond
    ((null list) nil)              ;список просмотрен до конца, элемент не найден
    ((equal el (car list)) t) ;очередная голова списка равна искомому, элемент найден
    (t (member el (cdr list)))))   ;если элемент не найден, продолжить поиск в хвосте
    списка
MEMBER
> (member 2 '(1 2 3))
T
> (member 22 '(1 2 3))
NIL
```

Примеры рекурсий…

Реверс списка (рекурсия по аргументу):

```
> (defun reverse (list)
    (cond
     ((null list) nil)      ;реверс пустого списка дает пустой список
     (t (append (reverse (cdr list)) (cons (car list) nil)))
     ;соединить реверсированный хвост списка и голову списка
    )
)
REVERSE

> (reverse '(one two three))
(THREE TWO ONE)

> (reverse ())
NIL
```

Примеры рекурсий…

Копирование списка (рекурсия по аргументу):

```
> (defun copy_list (list)
    (cond
      ((null list) nil)      ;копией пустого списка является пустой список
      (t (cons (car list) (copy_list (cdr list))))
;копией непустого списка является список, полученный из головы и
    ;копии хвоста исходного списка
    )
)
COPY_LIST

>(copy_list '(1 2 3))
(1 2 3)

>(copy_list ())
NIL
```

Другие виды рекурсии…

Рекурсию можно назвать простой, если в функции присутствует лишь один рекурсивный вызов. Такую рекурсию можно назвать еще рекурсией первого порядка. Но рекурсивный вызов может появляться в функции более, чем один раз. В таких случаях можно выделить следующие виды рекурсии:

- **параллельная рекурсия** – тело определения функции function_1 содержит вызов некоторой функции function_2, несколько аргументов которой являются рекурсивными вызовами функции function_1.

(defun function_1 … (function_2 … (function_1 …) … (function_1 …) …) …)

- **взаимная рекурсия** – в теле определения функции function_1 вызывается некоторая функции function_2, которая, в свою очередь, содержит вызов функции function_1.

(defun function_1 … (function_2 …) …)
(defun function_2 … (function_1 …) …)

- **рекурсия более высокого порядка** – в теле определения функции аргументом рекурсивного вызова является рекурсивный вызов.

(defun function_1 … (function_1 … (function_1 …) …) …

Параллельная рекурсия

Рассмотрим примеры параллельной рекурсии. В разделе, посвященном простой рекурсии, уже рассматривался пример копирования списка (функция copy_list), но эта функция не выполняет копирования элементов списка в случае, если они являются, в свою очередь также списками. Для записи функции, которая будет копировать список в глубину, придется воспользоваться параллельной рекурсией.

```
> (defun full_copy_list (list)
    (cond
;копией пустого списка является пустой список
      ((null list) nil)
;копией элемента-атома является элемент-атом
      ((atom list) list)
;копией непустого списка является список, полученный из копии головы
;и копии хвоста исходного списка
      (t (cons (full_copy_list (car list)) (full_copy_list (cdr list))))))
FULL_COPY_LIST
> (full_copy_list '(((1) 2) 3))
(((1) 2) 3)
> (full_copy_list ())
NIL
```

Взаимная рекурсия

Пример взаимной рекурсии – реверс списка. Так как рекурсия взаимная, в примере определены две функции: reverse и rearrange. Функция rearrange рекурсивна сама по себе.

```
> (defun reverse (list)
    (cond
      ((atom list) list)
      (t (rearrange list nil)))))
REVERSE

> (defun rearrange (list result)
    (cond
      ((null list) result)
      (t (rearrange (cdr list) (cons (reverse (car list)) result)))))
REARRANGE

> (reverse '(((1 2 3) 4 5) 6 7))
(7 6 (5 4 (3 2 1)))
```

Использование вспомогательных параметров.

В функциональном программировании избегают использовать глобальные переменные. Вместо этого стараются использовать параметры функций. Использование параметров вместо глобальных переменных приводит к более надежному и производительному методу программирования.

Накапливающий параметр. Использование накапливающего параметра рекурсивной функции позволяет создать контекст к серии вычислений, порождаемой рекурсивным вызовом. Такой подход является более корректным по сравнению с использованием некоторой глобальной переменной.

В процедурных языках программирования существует возможность использования промежуточных переменных, в которых можно сохранять промежуточные результаты. При решении задачи обращения списка можно применить процедурный подход, используя цикл DO и сохраняя промежуточные результаты в качестве значений вспомогательных переменных. В этом случае осуществляется перенос элементов исходного списка в результирующий список, который формируется с помощью CONS. Во вспомогательных переменных хранится хвост списка и формирующийся постепенно результат.

В функциональном программировании переменные таким образом не используются. Для этого используется вспомогательная функция, параметрами которой являются нужные вспомогательные переменные.

Тогда для функции обращения списка мы получим следующее определение:

```
(DEFUN rev1 (L)
        (per L nil))  =>  REV1
(DEFUN per (L rez)
        (COND ((NULL L) rez)
              (T (per (CDR L) (CONS (CAR L) rez)))))  =>  PER
(rev1 '(1 2 3 4))  =>  (4 3 2 1)
```

Вспомогательная функция PER рекурсивна по значению: она вырабатывает или непосредственный результат или осуществляет рекурсивный вызов.

На каждом шаге рекурсии очередной элемент из аргумента L переходит в аргумент REZ.

Обращенный список строится функцией CONS в аргументе REZ так же, как и в случае использования процедурного подхода.

Функции более высокого порядка.

Аргумент, значением которого является функция, называют **функциональным аргументом**. Функцию, имеющую функциональный аргумент, называют **функционалом**.

Функциональным аргументом может быть:

1) символьное имя, представляющее определение функции (системной или определенной пользователем);

2) безымянное лямбда-выражение;

3) замыкание.

Фактический параметр для функционального аргумента задается в виде формы, значением которой будет объект, который можно интерпретировать как функцию.

Получающий себя в качестве аргумента функционал называют **автоаппликативной функцией**. Функцию, возвращающую саму себя, **называют авторепликативной.**

Применяющие функционалы

- Функции, которые позволяют вызывать другие функции и применять функциональный аргумент к другим его аргументам называют применяющими функционалами.
- применяющие функционалы:
- APPLY **(APPLY fn список)**
- FUNCALL **(FUNCALL fn x1 x2 … xn)**

Примеры

```
(APPLY ‘+ ‘(4 5 6))
=> 15
(APPLY 'car  '((1 2 3)))
 => 1
(APPLY 'list '(1 2 3))
=> (1 2 3)
(DEFUN f (x)
     (COND ((NULL x) 'end)
          (T (PRINT (APPLY (CAR x) '(2 3))) (f (CDR x)))))
=> F
(f '(+ - * /))
=> 5
=> -1
=> 6
=> 2/3
=> END
```

Примеры

(FUNCALL ‘+ 4 5 6)
=> 15

(SETQ list ‘+)
=> +
(FUNCALL list 1 2)
=> 3
(LIST 1 2)
=> (1 2)

Отображающие функционалы

Отображающие или MAP-функционалы являются функциями, которые некоторым образом отображают список (последовательность) в новую последовательность или порождают побочный эффект, связанный с этой последовательностью.

Каждая из них имеет более двух аргументов, значением первого должно быть имя определенной ранее или базовой функции, или лямбда-выражение, вызываемое MAP-функцией итерационно, а остальные аргументы служат для задания аргументов на каждой итерации.

Количество аргументов в обращении к MAP-функции должно быть согласовано с предусмотренным количеством аргументов у аргумента-функции.

В общем случае вызов MAP-функций имеет вид:

(MAPx fn l1 l2 … ln),

где l1 l2 … ln – списки;

fn – функция от **n** аргументов.

Функция MAPCAR повторяет вычисление функции на элементах списка. Значение этой функции вычисляется путем применения функции fn к последовательным элементам xi списков, являющихся аргументами функции:

(MAPCAR fn ‘(x11 x12 ... x1m) …‘(xn1 xn2 ... xnm))

В качестве значения функционала возвращается список, построенный из результатов вызовов функционального аргумента MAPCAR.

Примеры.

```
(MAPCAR '+ '(1 2 3) '(1 2 3))
=> (2 4 6)
(MAPCAR '+ '(1 2 3) '(1 2 3)'(1 2 3))
=> (3 6 9)
(DEFUN f (x1 x2 x3) (list x1 x2 x3))
=> F
(mapcar 'f '(a b c) '(a b c)'(a b c))
=> (A A A) (B B B) (C C C))
(SETQ x '(a b c))
=> (A B C)
(MAPCAR 'cons x '((1) (2) (3)))
 => ((A 1) (B 2) (C 3))
> (mapcar #'(lambda (x) (+ 5 x)) '(1 2 3 4 5))
(6 7 8 9 10)
```

MAPLIST действует подобно MAPCAR, но действия осуществляет не над элементами списка, а над последовательными CDR этого списка, начиная с исходного:

(MAPLIST fn ‘(x11 x12 ... x1m) …‘(xn1 xn2 ... xnm))

Примеры.

(MAPLIST 'reverse '(1 2 3))

=> ((3 2 1) (3 2) (3))

(MAPLIST 'cons '(a b c) '(1 2 3))

=> (((A B C) 1 2 3) ((B C) 2 3) ((C) 3))

Функционалы MAPCAR и MAPLIST используются для программирования циклов специального вида и в определении других функций, поскольку с их помощью можно сократить запись повторяющихся вычислений.

Функции **MAPCAN** и **MAPCON** являются аналогами функций MAPCAR и MAPLIST. Отличие состоит в том, что MAPCAN и MAPCON не строят, используя LIST, новый список из результатов, а объединяют списки, являющиеся результатами, в один список.

Функциональные аргументы функций MAPCAN и MAPCON должны возвращать в качестве значения списки.

Примеры.

(MAPCAN 'f '(a b c) '(a b c)'(a b c))

=> (A A A B B B C C C)

(MAPCON 'reverse '(1 2 3))

=> (3 2 1 3 2 3)

Функции **MAPC** и **MAPL** также являются аналогами функций MAPCAR и MAPLIST, но отличаются тем, что не собирают и не объединяют результаты. Результаты просто теряются, а в качестве значения возвращается значение первого аргумента функции. Функции MAPC и MAPL являются псевдофункционалами, т.е. их используют для получения побочного эффекта.

Примеры.

```
(DEFUN f (x1 x2) (SET x1 x2))
=> F
(MAPC 'f  '(z1 z2 z3)  '(1 2 3))
=> (Z1 Z2 Z3)
 z2
=>  2
(MAPL 'PRINT '(1 2 3))
=> (1 2 3)
=> (2 3)
=> (3)
⇒    (1 2 3)
```

Вызовы функционалов можно объединять в более сложные структуры, так же, как и вызовы функций. Функционалы могут комбинироваться с использованием различных видов рекурсии.

http://lisp.ystok.ru/ru/lispworks/