Рекурсия

- Можно выделить следующие признаки классификации:
- – по количеству рекурсивных вызовов и их характеру;
- – по способу организации рекурсивных вычислений;
- – по характеру вычислительного процесса.

По количеству рекурсивных вызовов и их характеру

1. Простая рекурсия. Тело рекурсивной функции f содержит один рекурсивный вызов функции f.

(defun f(…)(… (f …) …).

2. Параллельная рекурсия. При параллельной рекурсии тело определения функции f содержит вызов некоторой функции g, несколько аргументов, которой является рекурсивными вызовами функции f.

(defun f(…)(… (g … (f …) (f …) …) …)).

3. Взаимная (косвенная) рекурсия. При взаимной рекурсии тело определения функции f содержит вызов некоторой другой функции g, которая в свою очередь содержит вызов функции f.

(defun f(…)(… (g …) …),

(defun g(…)(… (f …) …).

4. Рекурсия высокого порядка. Рекурсия более высокого порядка возникает тогда, когда тело определения функции f содержит рекурсивный вызов функции f, аргументом которого является рекурсивный вызов f:

(defun f(…) … (f … (f …) …) …).

По способу организации рекурсивных вычислений

- С одной (несколькими) терминальными ветвями и одним рекурсивным вызовом.
- Пополняющая рекурсия или рекурсия с отложенными вычислениями.
- Рекурсия с накоплением по условию.
- «Хвостовая» рекурсия
- «CAR/CDR» рекурсия

С одной (несколькими) терминальными ветвями и одним рекурсивным вызовом

```
(defun fun(x)
   (cond (end-test-1 end-value-1)
          …
         (end-test-n end-value-n)
         (T (fun reduced-x))
   )
)
```

С одной (несколькими) терминальными ветвями и одним рекурсивным вызовом

```
(defun find-first-atom(x)
   (cond ((atom (car x)) (car x))
         (t (find-first-atom (cdr x)))
   )
)
(defun anyoddp(x)
   (cond ((null x) nil)
         ((oddp (car x)) t)
         (t (anyoddp (cdr x)))
   )
)
```

Пополняющая рекурсия или рекурсия с отложенными вычислениями

```
(defun fun (X)
   (cond (end-test end-value)
         (T (aug-fun   aug-val
                       (fun reduced-x)
            )
         )
   )
)
```

Пополняющая рекурсия или рекурсия с отложенными вычислениями

```
(defun sum(n)
    (cond ((= n 1) 1)
          (t (+ n (sum (- n 1))))
    )
)
(defun laugh(n)
    (cond ((zerop n) nil)
          (t (cons “ha” (laugh (- n 1))))
    )
)
```

Длина списка

```
(defun _length(ls) (
  cond ((null ls) 0)
          (t (+ 1 (_length (cdr ls))))
  )
)
```

Рекурсия с накоплением по условию

```
(defun fun(x)
    (cond (end-test end-value)
          (aug-test (aug-fun aug-val
                             (fun reduced-x)
          )
          (T (fun reduced-x))
    )
)
```

Рекурсия с накоплением по условию

```
(defun extract-symbols(x)
   (cond ((null x) nil)
         ((symbolp (car x))
             (cons (car x)
                   (extract-symbols (cdr x))
             )
         )
         (t (extract-symbols (cdr x)))
   )
)
```

«Хвостовая» рекурсия

```
(defun extract-symbols(x) (labels
  ((inner(out in) (cond
     ((null in) out)
     ((symbolp (car in))
        (inner (append out (list (car in))) (cdr in)))
     (t (inner out (cdr in)))
   )))
   (inner nil x)
   )
)
```

«Хвостовая» рекурсия

```
(defun inner(out in)
  (cond
     ((null in) out)
     ((symbolp (car in))
        (inner (append out (list (car
  in))) (cdr in)))
     (t (inner out (cdr in)))
   )
)
```

«Хвостовая» рекурсия. Факториал

```
(defun fact(n result)
     (if (<= n 1) result
                  (fact (- n 1)
                        (* n result)
                  )
     )
)
```

CAR/CDR рекурсия

```
(defun fun (X)
   (cond (end-test-1 end-value-1)
         (end-test-2 end-value-2)
         (T (combiner (fun (CAR X))
                      (fun (CDR X))
            )
         )
   )
)
```

CAR/CDR рекурсия

```
(defun find-number (x)
   (cond ((numberp x) x)
         ((atom x) nil)
         (t (or (find-number (car x))
                (find-number (cdr x)))
         )
   )
)
```

CAR/CDR рекурсия

```
(defun extract-symbols (x)
   (cond ((null x) nil)
         ((symbolp (car x)) (cons (car x)
                                  (extract-symbols (cdr x))
             )
         )
         ((listp (car x))
             (cons (extract-symbols (car x))
                   (extract-symbols (cdr x))
             )
         )
         (t (extract-symbols (cdr x)))
   )
)
```

Еще примеры рекурсии

Задача: построить функцию, которая находит сумму элементов массива

Еще примеры рекурсии

Задача: построить функцию, которая находит сумму элементов массива

1 способ

```
(defun sum1(a i)
  (if (>= i (array-dimension a 0))
      0
      (+ (aref a i) (sum1 a (+ i 1))))
  )
)
```

Еще примеры рекурсии

Задача: построить функцию, которая находит сумму элементов массива

2 способ

```
(defun sum2(a &optional (i 0))
  (if (>= i (array-dimension a 0))
      0
      (+ (aref a i) (sum2 a (+ i 1))))
  )
)
```

Еще примеры рекурсии

Задача: построить функцию, которая находит сумму элементов массива

3 способ

```
(defun sum3(a &aux (n (array-dimension a 0)))
   (labels ((inner(i sum)
              (if (>= i n)
                  sum
                  (inner (+ i 1) (+ sum (aref a i)))
              )
            ))
   (inner 0 0)
   )
)
```

Еще примеры рекурсии

Задача: элемент ряда Фибоначчи