ЛЕКЦИЯ 2
Рекурсия и итерация. Хвостовая рекурсия

Что было раньше

Подстановочная модель (аппликативный п. в.). Правила:

1. Если выражение – литерал, то вернуть само выражение.
2. Если выражение – имя, вернуть его значение в текущем окружении.
3. Если выражение – спец. форма, то вычислить его по правилам этой спец. формы.
4. Если выражение – комбинация, то:
   4.1. Вычислить его подвыражения в произвольном порядке.
   4.2. Если первое подвыражение – встроенная процедура, то применить её к операндам.
   4.3. Если оно – пользовательская функция, подставить в её тело значения аргументов и вычислить полученное выражение.

Функции и процессы

факториал

```
(define (fact1 n)
          (if (< n 2) 1
             (* n (fact1 (- n 1)))))
```

Процесс, порождаемый > (fact1 3) -->

(if (< 3 2) 1 (3 (fact1 (- 3 1)))) -->*
(3 (fact1 2)) -->*
(3 (if (< 2 2) 1 (* 2 (fact1 (- 2 1))))) -->*
(3 (* 2 (fact1 1))) -->*
(3 (* 2 (if (< 1 2) 1 (* 1 (fact1 (- 1 1)))))) -->*

расширение

(3 (* 2 1)) -->*
(3 2)*

сжатие

-> 6

Функции и процессы

> (fact1 6) -->

(6 (fact1 5)) -->*

(6 (* 5 (fact1 4))) -->*

(6 (* 5 (* 4 (fact1 3)))) -->*

(6 (* 5 (* 4 (* 3 (fact1 2))))) --> …*

отложенные вычисления

Количество шагов *линейно*. Память *линейна*.

Как и авторы учебника мы считаем сложность не от длины записи параметра, а от значения параметра.

Функции и процессы

Итеративный факториал

```
(define (fact2 n)
    (define (loop i result)
        (if (< i 2) result
            (loop (- i 1) (* i result))))
    (loop n 1))
```

Процесс порождаемый > (fact2 3) -->

(loop 3 1) -->
(if (< 3 2) 1 (loop (- 3 1) (3 1))) -->*
(loop 2 3) -->
(if (< 2 2) 3 (loop (- 2 1) (2 3))) -->*
(loop 1 6) -->
(if (< 1 2) 6 (loop (- 1 1) (1 6)))*
-> 6

Функции и процессы

> (fact2 6) -->
(loop 6 1) -->
(loop 5 6) -->
(loop 4 30) -->
(loop 3 120) -->
(loop 2 360) -->
(loop 1 720)
-> 720

Отложенных вычислений нет!

Ситуация, когда рекурсивный вызов завершает вычисление (не имеет связанных с ним отложенных вычислений) – это *хвостовая рекурсия.* Этот вызов можно вычислять в том же стековом кадре, что и вызов, породивший его. Количество шагов *линейно*. Память *постоянна*.

Виды рекурсии

Описания факториала:

• обычная (*не хвостовая*) линейная рекурсия:

$0! = 1, 1! = 1, n! = n*(n-1)!$

• *хвостовая* линейная рекурсия:

$n! = D(n, 1), D(0, i) = i, D(1, i)= i, D(j, i)=D(j-1, j*i)$

Далее все $F_j(n)$ – это разные способы описать числа Фибоначчи.

• *множественная* рекурсия:

$F_1(0) = 0, F_1(1) = 1, F_1(n) = F_1(n-1) + F_1(n-2)$ *или* $F_1(n) = F_1(n-2)+…+ F_1(0)+1$

• *взаимная* рекурсия:

$F_2(n) = A(n) + B(n), A(1) = 0, B(1) = 1, A(n) = A(n-1) + B(n-1), B(n) = A(n-1)$

• вложенная рекурсия:

$F_3(n) = C(n, 0), C(1, i) = 1 + i, C(2, i) = 1 + i, C(n, i) = C(n-1, i + C(n-2, 0))$

Рекурсивные и итеративные процессы

- Если в ходе процесса возникает цепочка отложенных вычислений, то процесс рекурсивный.
- Если в ходе процесса отложенных вычислений нет, то процесс итеративный.
- Рекурсивный процесс, в котором число шагов линейно – линейно рекурсивный.
- Итеративный процесс с линейным количеством шагов – линейно итеративный.
- Описание функции рекурсивно, но порождаемые ей процессы могут быть рекурсивны либо итеративны, в зависимости от описания.

Математики сложность анализируют как зависимость от *длины* записи входа. Мы, вслед за авторами учебника, будем определять порядок от *значения* параметра n. Так, n! считается линейно, не экспоненциально.

Нелинейный рекурсивный процесс

- Рекурсивное вычисление чисел Фибоначчи

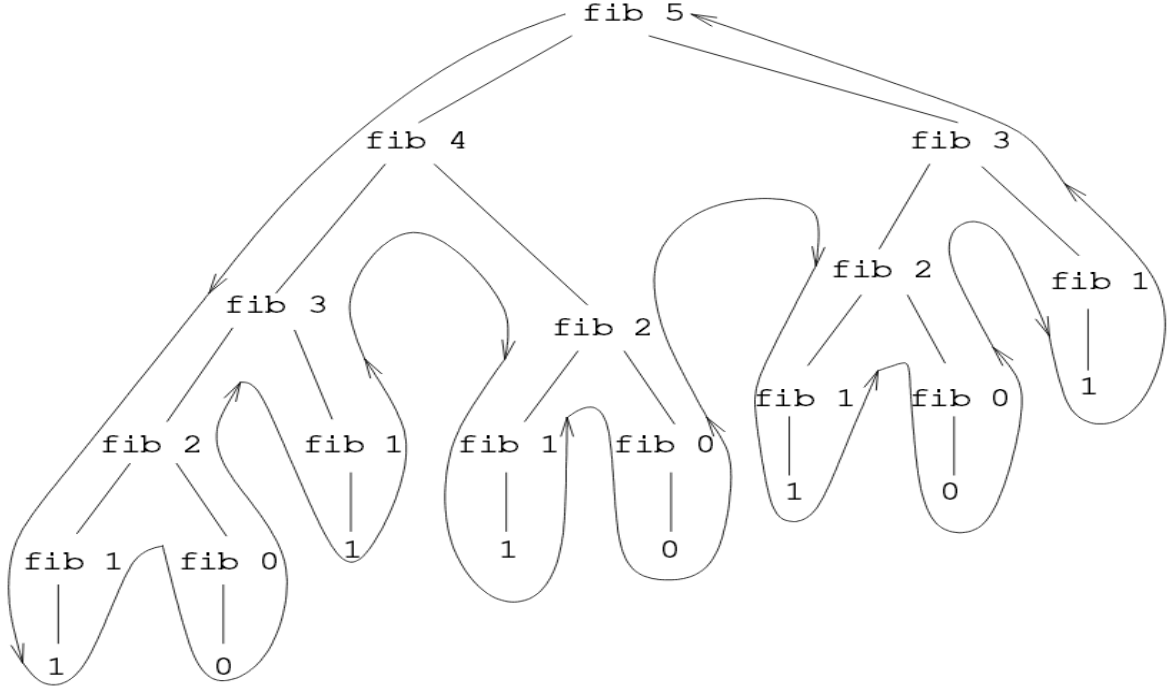
```
(define (fib n)
    (cond ((= n 0) 0)
          ((= n 1) 1)
          (else (+ (fib (- n 1)) (fib (- n 2))))))
```

- Вычисление > (fib 5) -->

(fib 5) -->
(+ (fib 4) (fib 3)) -->
(+ (fib 4) (+ (fib 2) (fib 1))) -->
(+ (fib 4) (+ (+ (fib 1) (fib 0)) 1)) -->
(+ (fib 4) (+ 1 1)) -->
(+ (fib 4) 2) -->
(+ (+ (fib 3) (fib 2)) 2) --> ...
-> 5

Нелинейный рекурсивный процесс

Дерево рекурсивного вычисления чисел Фибоначчи



Количество шагов *экспоненциально*. Память *линейна.*

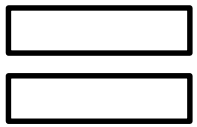
Линейна ли памят?

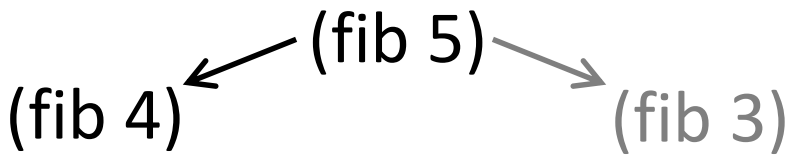
Почему в рекурсивном процессе вычисления чисел Фибоначчи память линейна?

(fib 5)

Линейна ли память?

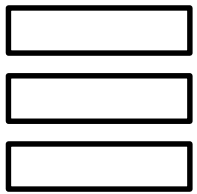
Почему в рекурсивном процессе вычисления чисел Фибоначчи память линейна?

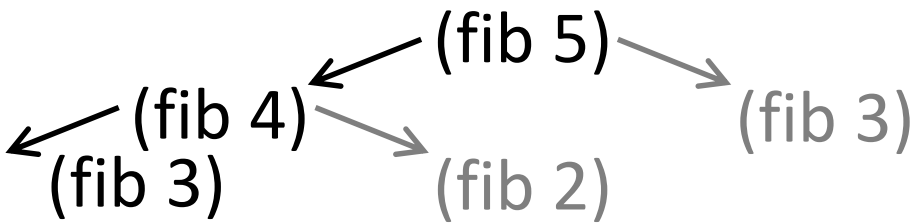




Линейна ли памят?

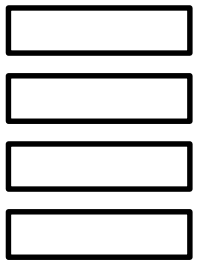
Почему в рекурсивном процессе вычисления чисел Фибоначчи память линейна?

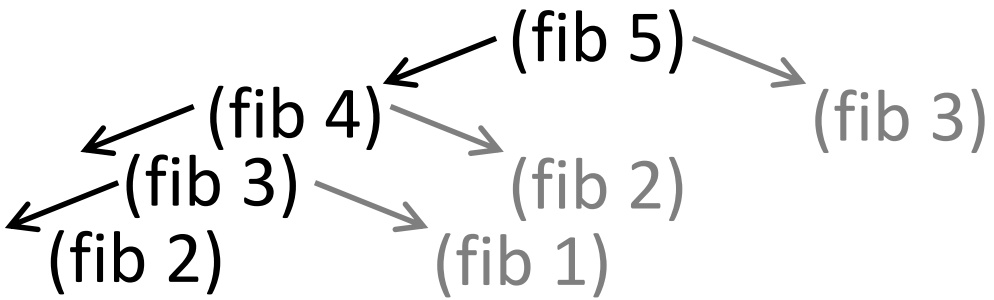




Линейна ли памят?

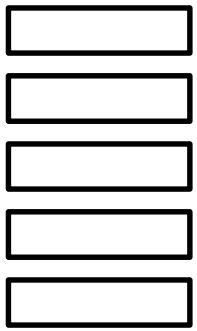
Почему в рекурсивном процессе вычисления чисел Фибоначчи память линейна?

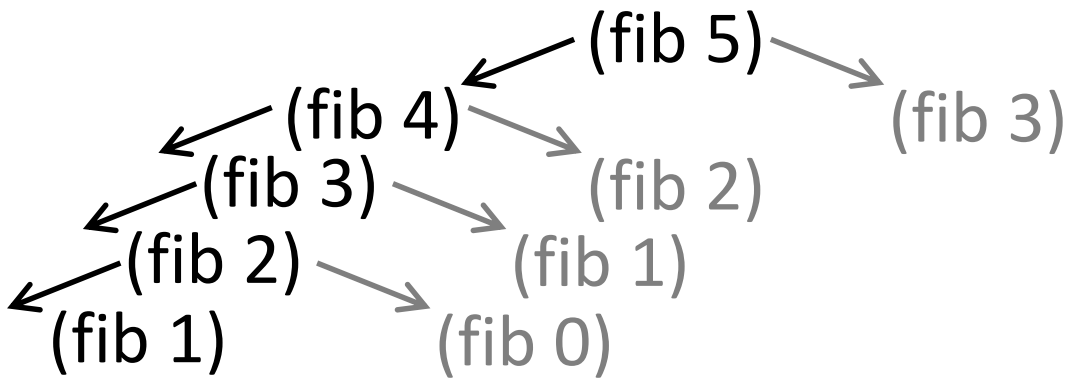




Линейна ли памят?

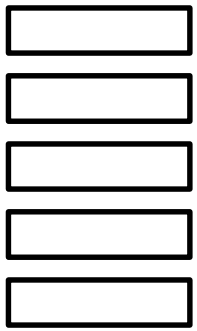
Почему в рекурсивном процессе вычисления чисел Фибоначчи память линейна?

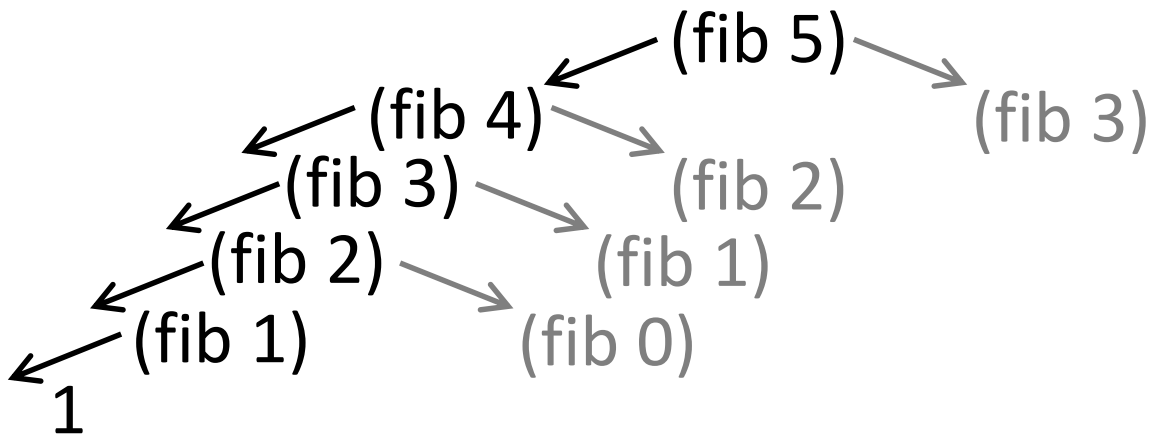




Линейна ли памят?

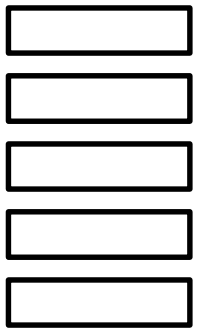
Почему в рекурсивном процессе вычисления чисел Фибоначчи память линейна?

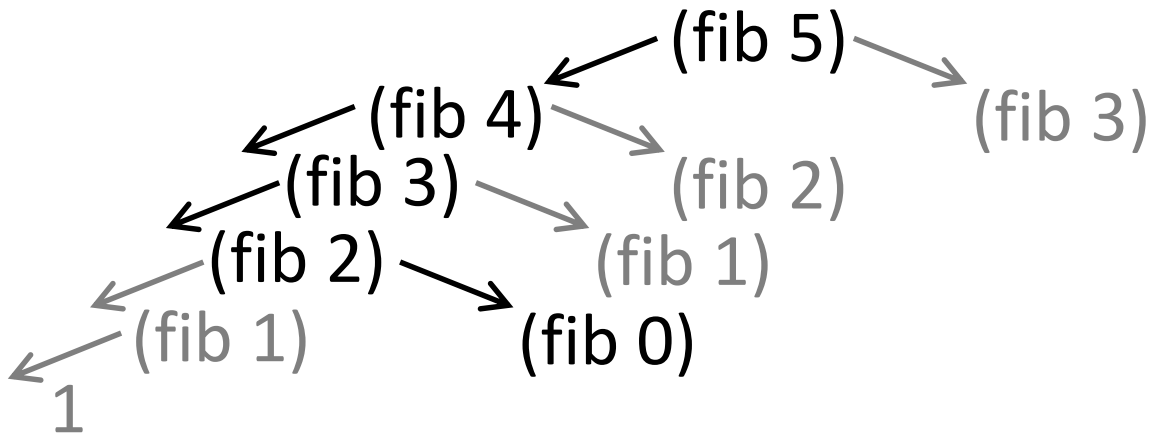




Линейна ли памят?

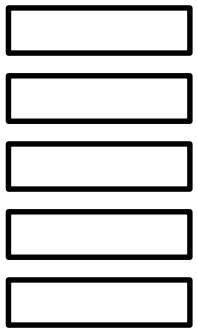
Почему в рекурсивном процессе вычисления чисел Фибоначчи память линейна?

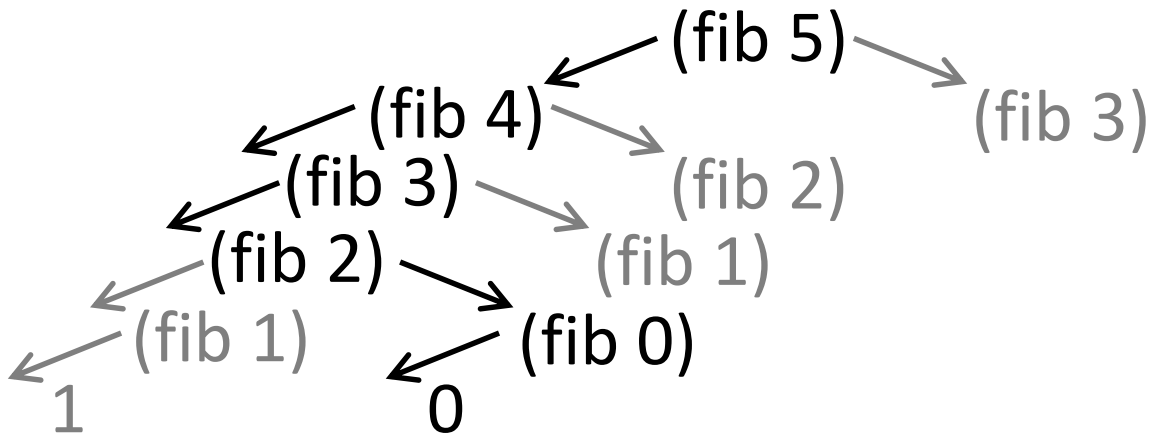




Линейна ли память?

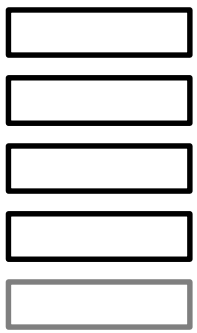
Почему в рекурсивном процессе вычисления чисел Фибоначчи память линейна?

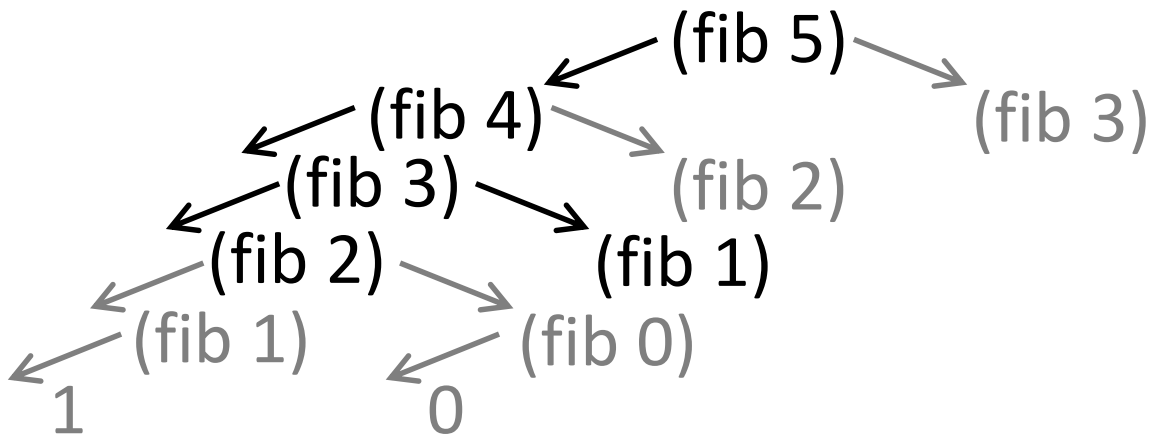




Линейна ли память?

Почему в рекурсивном процессе вычисления чисел Фибоначчи память линейна?





Итеративное вычисление чисел Фибоначчи

```
(define (fib2 n)
    (define (loop i fib-n-1 fib-n-2)
        (if (= i 0) fib-n-2
                    (loop (- i 1) (+ fib-n-1 fib-n-2) fib-n-1)))
    (loop n 1 0))
```

> (fib2 5) -->
(loop 5 1 0) -->
(loop 4 1 1) -->
(loop 3 2 1) -->
(loop 2 3 2) -->
(loop 1 5 3) -->
(loop 0 8 5)
-> 5

Количество шагов линейно, память константа.

Вспомним оборачивание списка

```
(define (my-reverse lst)
    (if  (null? lst) '()
                  (append (my-reverse (cdr lst)) (list (car lst)))))
```

> (my-reverse '(1 2 3)) -->

(append (my-reverse '(2 3)) (list 1)) -->

(append (append (my-reverse '(3)) (list 2)) '(1)) -->

(append (append (append (my-reverse '()) (list 3)) '(2)) '(1)) -->

(append (append (append '() '(3)) '(2)) '(1)) -->

(append (append '(3) '(2)) '(1)) -->

(append '(3 2) '(1))

-> (3 2 1)

Отложенные операции! Рекурсивный процесс.

Итеративное оборачивание списка

```
(define (reverse2 lst)
    (define (loop lst result)
        (if (null? lst) result
            (loop (cdr lst) (cons (car lst) result))))
    (loop lst '()))
```

> (reverse2 '(1 2 3)) -->

(loop '(1 2 3) '()) -->

(loop '(2 3) '(1)) -->

(loop '(3) '(2 1)) -->

(loop '() '(3 2 1))

-> (3 2 1)

Линейно итеративный процесс (параметр – длина списка)

Возведение в степень

- $a^n = a \cdot a \cdots a = a \cdot a^{n-1}$

```
(define (my-expt a n)
    (if (= n 0)
        1
        (* a (my-expt a (- n 1)))))
```

> (my-expt 10 4) -->

(if (= 4 0) 1 (10 (my-expt 10 (- 4 1)))) --> …*

есть отложенные операции!

Количество шагов линейно. Память линейна

Итеративное возведение в степень

- перепишем с накоплением

```
(define (my-expt1 a n)
   (define (loop i result)
     (if (= i 0)       result
                       (loop (- i 1) (* a result))))
   (loop n 1))
> (my-expt1 10 4) -->
(loop 4 1) -->
(loop (- 4 1) (* 10 1)) -->
(loop (- 3 1) (* 10 10)) -->
(loop (- 2 1) (* 10 100)) --> ...
-> 10000
```

отложенные операции отсутствуют!

Количество шагов линейно. Память постоянна

Нелинейное итеративное возведение в степень

```
(define (my-expt2 a n)
   (define (loop a i result)
     (cond ((= i 0) result)
           ((even? i) (loop (* a a) (/ i 2) result))
           (else (loop a (- i 1) (* a result)))))
   (loop a n 1))
```

> (my-expt2 10 4) -->

(loop 10 4 1) -->

(loop (10 10) (/ 4 2) 1) -->*

(loop (100 100) (/ 2 1) 1) -->*

(loop 10000 1 1)

-> 10000

Количество шагов O(log n). Память постоянна

Итеративный НОД с логарифмическим ростом

```
(define (my-gcd a b)
   (if (= b 0) a
          (my-gcd b (remainder a b))))
```

> (my-gcd 4 12) -->

(my-gcd 12 4) -->

(my-gcd 4 0)

-> 4

Количество шагов O(log n). Память постоянна

Именованный let

```
(define (my-expt2 a n)
   (define (loop a i result)
     (cond ((= i 0) result)
           ((even? i) (loop (* a a) (/ i 2) result))
           (else (loop a (- i 1) (* a result)))))
   (loop a n 1))
```

Можно переписать без явного вложенного define

```
(define (my-expt3 a n)
   (let loop ((a a) (i n) (result 1))
     (cond ((= i 0) result)
           ((even? i) (loop (* a a) (/ i 2) result))
           (else (loop a (- i 1) (* a result)))))
)
```

Именованный let

(let <вспомогательноеИмя>
((<var_1> <$expr_1$>)
(<var_2> <$expr_2$>)
…
(<var_N> <$expr_N$>))
<тело>); в теле может быть использован вызов функции вспом-Имя!

Конструкция аналогична следующему коду

(define (<вспомогательноеИмя> <var_1> <var_2> … <var_N>) <тело>)
(<вспомогательноеИмя> <$expr_1$> <$expr_2$> … <$expr_N$>)

или

(define <вспомогательноеИмя>
(lambda (<var_1> <var_2> … <var_N>) <тело>))
(<вспомогательноеИмя> <$expr_1$> <$expr_2$> … <$expr_N$>)

Именованный let

```
(define (reverse2 lst)
        (define (loop lst result)
                (if (null? lst) result
                        (loop (cdr lst) (cons (car lst) result))))
        (loop lst '()))
```

Перепишем с именованным let

```
(define (reverse3 lst)
        (let loop ((lst lst) (result '()))
                (if (null? lst) result
                        (loop (cdr lst) (cons (car lst) result))))
)
```

Именованный let

```
(define (fib2 n)
        (define (loop i fib-n-1 fib-n-2)
                (if (= i 0) fib-n-2
                        (loop (- i 1) (+ fib-n-1 fib-n-2) fib-n-1)))
        (loop n 1 0))
```

Перепишем с именованным let

```
(define (fib3 n)
        (let loop ((i n) (fib-n-1 1) (fib-n-2 0))
                (if (= i 0) fib-n-2
                        (loop (- i 1) (+ fib-n-1 fib-n-2) fib-n-1)))
)
```

Именованный let

```
(define (fact2 n)
        (define (loop i result)
                (if (< i 2) result
                        (loop (- i 1) (* i result))))
        (loop n 1))
```

Перепишем с именованным let

```
(define (fact3 n)
        (let loop ((i n) (result 1))
                (if (< i 2) result
                        (loop (- i 1) (* i result))))
)
```

Рекомендация по стилю: использовать именованный let *или* define *для локальных функций,* let *и* let* *для локальных имён.*

Итоги лекции 2

- Рекурсивное описание функции может породить рекурсивный процесс или итеративный процесс.
- Писать программу надо, оценивая сложность по шагам и по памяти.
- При осуществлении итеративных процессов память используется экономно (по сравнению с рекурсивными процессами).
  Т. е. функциональные программы, порождающие итеративные процессы, по эффективности аналогичны императивным программам с циклами.
- Запрограммировать итеративную реализацию можно
  либо с помощью вспомогательной функции с доп. параметрами,
  либо с помощью именованного let.